NANYANG TECHNOLOGICAL UNIVERSITY

A SEMANTIC-BASED ANALYSIS OF ANDROID MALWARE FOR DETECTION, GENERATION, AND TREND ANALYSIS

GUOZHU MENG

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

2017

THESIS ABSTACT

A Semantic-based Analysis of Android Malware for Detection, Generation, and Trend Analysis

by

GUOZHU MENG

Doctor of Philosophy
School of Computer Science and Engineering
Nanyang Technological University, Singapore

Android has grown to be the most popular mobile operating system since its release in 2008. Due to its openness and ease of use, it attracts thousands of vendors and developers working on Android application development. Millions of apps provide a variety of functionalities to Android users, such as online shopping, instant messaging, gaming and map service. However, Android becomes a hot attack target of cybercriminals due to its prevalence. According to the security report of Symantec in 2016, the number of Android malware has reached 13 million in 2015. Android malware is uploaded into either Google official market or unofficial markets everyday by cybercriminals which put users under a high risk. The malware may steal users' sensitive information, elevate the privilege, remote control devices, and encrypt users' files for ransom. It is non-trivial to understand the risks and develop effective mitigation against them.

Malware is the critical and non-trivial issue in Android security. In order to prevent malware from attacking the users, we need a better understanding of Android malware and its behaviors, which can facilitate the extraction of representative features from malware, and thereby enhance malware detection. The malware and anti-malware tools are keeping evolving during the process of competition. Therefore, it is valuable to learn the characteristics of evolving malware, and weakness of existing anti-malware tools. Moreover, a sustaining malware analysis and security assessment is lacking for the Android world.

In order to address these problems, we propose a semantic based malware analysis on these topics with the following achievements in this thesis:

1. We propose a precise semantic model of Android malware based on Deterministic Symbolic Automaton (DSA) for the purpose of malware comprehension, detection and classification. Based on DSA, we develop an automatic analysis framework, named SMART, which learns DSA by detecting and summarizing semantic clones from malware families, and then extracts semantic features from the learned DSA to classify malware according to the attack patterns. We conduct the experiments in both malware benchmark and 223,170 real-world apps. The results show that SMART builds meaningful semantic models and outperforms both state-of-the-art approaches and anti-virus tools in malware detection. SMART identifies 4583 new malware in real-world apps that are missed by most anti-virus tools. The classification step further identifies new malware variants and unknown families.

2. We first propose a meta model for Android malware to capture the common attack features and evasion features in the malware. Based on this model, we develop a framework, MYSTIQUE, to automatically generate malware covering four attack features and two evasion features, by adopting the software product line engineering approach. With the help of MYSTIQUE, we conduct experiments to 1) understand Android malware and the associated attack features as well as evasion techniques; 2) evaluate and compare the 57 off-the-shelf anti-malware tools, 9 academic solutions and 4 Android market vetting processes in terms of accuracy in detecting attack features and capability in addressing evasion. Last but not least, we provide a benchmark of Android malware with proper labeling of contained attack and evasion features. Moreover, we extend this work to MYSTIQUE-S to explore the capabilities of anti-malware tools detecting malware with dynamic code loading. MYSTIQUE-S automatically selects attack features under various user scenarios and delivers the corresponding malicious payloads at runtime. Relying on dynamic code binding (via service) and loading (via reflection) techniques, MYSTIQUE-S enables the dynamic execution of payloads on user devices at runtime. Experimental results on real-world devices show that existing Anti-Malware Tools (AMTs) are incapable of detecting most of our generated malware. Last, we propose some enhancements for existing anti-malware tools.

3. We propose a systematic approach to study Android malware, unveil security issues, obtain insightful conclusions and highlights, and predict the future trend for research. We have collected 4,267,178 Android apps from a variety of Android marketplaces, where 1,004,550 malware variants are identified and analyzed. Different from previous works, this work focuses on the differences and evolution of apps' characteristics, and identifies multiple security-related issues concerned by both academia and industry. In order to provide a comprehensive view for these issues, we propose four analyses on individual app, malware family, malware author, and market, to conduct our study and guide the analysis. Furthermore, we propose six dimensions to cluster apps for different analysis tasks to achieve efficiency and accuracy in the large-scale analysis. Some of the key findings reflect the characteristics of attacks, and the weaknesses in protection, which can benefit all stakeholders.

Contents

List of Figures

List of Tables

1

Introduction

Android has become a popular mobile operating system deployed on more than 1 billion devices [2]. A tremendous number of Android apps are developed, providing a variety of functionalities for users. Almost all activities can be done in mobile devices. Along with the popularity of mobile apps, attackers start to swarm into this area, and attempt to make profits or damages. However, Android has grown to be the most popular mobile operating system since its release in 2008. Due to its openality and ease of use, it attracts thousands of vendors and developers working on Android application development. Millions of apps provide a variety of functionalities to Android users, such as online shopping, instant messaging, gaming and map service. However, Android becomes a hot attack target due to its prevalence. Android malware is uploaded into either Google official market or unofficial markets everyday by cybercriminals which put users under a high risk. The malware may steal users' sensitive information, elevate the

privilege, remote control devices, and encrypt users' files for ransom. It is non-trivial to understand our risks and the mitigation against them.

1.1 Motivations and Goals

Android users are facing increasing threats from a tremendous amount of malware, and about 30.6% of analyzed apps by SYMANTEC are classified as malware [3] in 2015. Meanwhile, there have been emerging many kinds of approaches and techniques to understand, detect, and classify Android malware. However, Android malware is keeping evolved by employing more evasive techniques, exploiting zero-day vulnerabilities, and attacking new targets (see Chapter 2.2); an effective malware detection is demanding due to such huge number of Android apps. To understand the state affairs of security in Android ecosystem, we have investigated many works on malware detection and Android protection [4]. It is concluded that the Android world is still facing many security problems and challenges with regard to the increasing number of malware. In particular, according to the investigation of Android ecosystem, we identify the following problems to be answered in this thesis:

- **P1. The semantic understanding of Android malware.** The essence of Android malware can depict the fundamental differences between Android malware and benign apps, and meanwhile cluster original malware and its variants. It is the first and fundamental step to understand malware, and thereby conduct the further research works.
- **P2. Effective Android malware detection.** There are millions of Android apps providing convenient functionalities for users. However, a large portion of them are designed maliciously and will cause considerate damages to users. Therefore, it becomes very necessary and demanding to design an accurate and scalable malware detection approach to rapidly check the apps in real time.
- **P3. The evaluation of existing anti-malware tools.** The unveiling of the weakness and shortcomings of existing anti-malware tools can help to improve their

detection capabilities and quality. Therefore, it is significant to understand the capabilities of anti-malware tools detecting a variety of malware, and the resistance to evasion techniques such as obfuscation and dynamic code loading.

- **P4. The emergence, characteristics, and evolution of Android malware.** It is desiderate to answer the important questions in Android world, such as how Android apps become malware, what are the common characteristics of Android malware, how they evolve, and how secure of Android markets. The answers to these questions facilitate both the proactive protection and the retroactive protection. Therefore, it is useful to depict a panoramic view of Android malware.

These problems motivate the main work in the thesis. It is non-trivial to solve the aforementioned problems since there exist many technical challenges. We list the following challenges for each problem, respectively:

- **C1.1. Sematic modelling of malware**. A concise and precise model of Android malware facilitates the detection. The model should not only unveil the essence of malicious behaviors in Android malware, but also summarize the evolutionary changes in its variants.

- **C1.2. Extraction of malicious behavior**. Generally, malicious behaviors may lurk in harmless code. A large portion of Android malware is created by inserting malicious code into benign apps [5]. Therefore, it is non-trivial to extract the exact malicious behaviors in amounts of code.

- **C2. Scalable and accurate detection**. Thousands of Android apps are uploaded into Android markets every day, which contain a large number of malware [3]. It is desired to design a scalable and accurate approach of malware detection since there are millions of Android apps on use in which a huge risk exists and may threaten users' normal life.

- **C3.1. Quality assessment of generated malware**. The work in Chapter 4 and 5 generates a large amount of Android malware to audit anti-malware tools. However, we need some goals to guide automated malware generation. The criteria

to assess the quality of generated malware can guide the malware generation process, and thereby facilitate the auditing of anti-malware tools with keep-evolving malware.

- **C3.2. Validity of generated malware**. As the malware is generated according to the semantic model, we attempt to prove their maliciousness — whether malicious behaviors can be triggered and carried out on real Android devices.

- **C4.1. High dimensionality of Android malware**. The analyzed dataset contains 3 million benign apps and 1 million malicious apps. The huge number creates unique features and raises the difficulty of analysis. In order to obtain meaningful and useful conclusions from the big data, we have to resolve accumulated noise, handle biased data, and purify the analyzed data.

- **C4.2. Effective approach to acquire security conclusions**. Most of existing approaches analyze Android apps individually and unilaterally, while neglecting the correlation and integrity of Android apps. Therefore, it is non-trivial to propose an effective approach to obtain security conclusions from the big dataset.

- **C4.3. Utilities of security conclusions**. Practicability is always one of the important factors in security domain. The acquired security conclusions should be efficiently applied into the real world, for example, improving the existing anti-malware tools, enhancing the security of Android system, and pre-protection of Android markets. Therefore, it is challenging to represent, visualize, and pragmatize these security conclusions.

1.2 Main Works and Contributions

In the following, four main works are conducted to solve the aforementioned problems and technical challenges as shown in Fig. 1.1. First, we take Android malware as input and build a sematic model for Android malware. The model is used to extract features from Android apps to check, and detect other malware and variants. In addition, the model is used to automatically generate Android malware to evaluate anti-malware

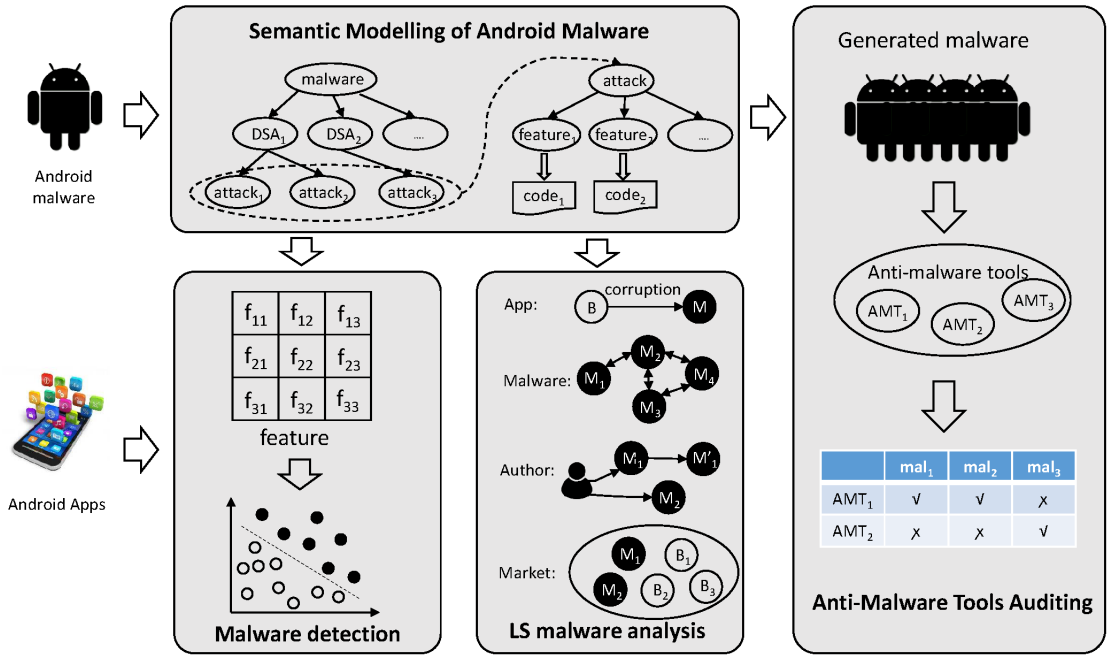

FIGURE 1.1: The overview of the thesis

tools. Last, we conduct a large-scale analysis on Android apps by leveraging the semantic model to conclude security insights in different dimensions. We elaborate these works as follows:

1. Semantic Modelling of Android Malware. A better model for Android malware can facilitate the comprehension of malware evolution, the detection of Android malware, and the evaluation of Anti-Malware tools, and these three works construct the majority of this thesis. As the basis of the thesis, we conduct a work to extract the essence of Android malware, and build a semantic model for ease of use. In order to identify essence of Android malware, we perform a difference analysis on existing Android malware. The malware in DREBIN are well labelled and categorized, and malware in the same family share the common malicious behaviors. We perform a clone detection at method level on these malware samples, and identify the common methods which have high similarity in between. In addition, we use Deterministic Symbolic Automaton (DSA) to summarize the clone methods as the essence of Android malware. Since DSA can present both the commonality and differences between Android malware, we can learn the diversity and the evolution from it. The features that are extracted from DSA can better describe and depict Android malware, and thereby lead a better detection result. Moreover, the different combinations of these features can

construct different Android malware which can be used to evaluate the performance of Anti-Malware tools.

Contributions:

- We propose a hierarchical semantic model of Android malware based on DSA, which links malware family, malicious behaviors and fine-grained actions. To the best of our knowledge, this is the first work in this direction.
- We propose an effective method to extract the common malicious behaviors as DSA from malware variants of an Android malware family.

2. Scalable and Accurate Malware Detection. Based on the proposed semantic model of Android malware, we develop an automatic analysis framework, named Semantic Modelling of AndRoid ATtack (SMART), which learns DSA by detecting and summarizing semantic clones from malware families as aforementioned. DSA is used to feature Android malware, and classify them into a specific family. We conduct the experiments in both malware benchmark and 223,170 real-world apps. The results show that SMART builds meaningful semantic models and outperforms both state-of-the-art approaches and anti-virus tools in malware detection. SMART identifies 4,583 new malware in real-world apps that are missed by most anti-virus tools. The classification step further identifies new malware variants and unknown families.

Contributions:

- We combine machine learning and static analysis to detect and classify malware. The first phase quickly identifies suspicious apps, and the second phase uses DSA inclusion to further confirm and classify the suspicious apps. Hence we can achieve both scalability and accuracy.
- We test SMART on 5,560 known malware and 223,170 real-world apps. The results show that SMART outperforms many anti-virus (AV) tools and academic approaches, with a precision of 86.7% for malware detection. For wild prediction, it detects 4,583 new malware variants.

3. Auditing Anti-Malware Tools. The semantic model of Android malware inspires us to conduct a work to evaluate the performance of anti-malware tools (AMTs). Existing anti-malware tools have many limitations: 1) the figure or signature based detection mechanism is susceptible for malware variants; 2) the detection performance is greatly affected by evasion techniques such as obfuscation; 3) new malware and attacks that anti-malware tools rarely detect are pervasive and disturb our normal life. Therefore, we propose a meta model for Android malware to capture the common attack features (AFs) and evasion features (EFs). Based on this model, we develop a framework, MYSTIQUE, to automatically generate malware covering four attack features and two evasion features, by adopting the software product line engineering approach. With the help of MYSTIQUE, we conduct experiments to 1) understand Android malware and the associated attack features as well as evasion techniques; 2) evaluate and compare the 57 off-the-shelf anti-malware tools, 9 academic solutions and 4 Android market vetting processes in terms of accuracy in detecting attack features and capability in addressing evasion. Last but not least, we provide a benchmark of Android malware with proper labeling of contained attack and evasion features.

In addition, we extend this work to MYSTIQUE-S which employs dynamic code generation and loading techniques to produce malware so that we can audit the AMTs at runtime. MYSTIQUE-S is a service oriented malware generation system. It automatically selects attack features under various user scenarios and delivers the corresponding malicious payloads at runtime. Relying on dynamic code binding (via service) and loading (via reflection) techniques, MYSTIQUE-S enables dynamic execution of payloads on user devices at runtime. Experimental results on real-world devices show that existing AMTs are incapable of detecting most of our generated malware. Last, we propose the enhancements for existing AMTs.

Contributions:

- We recognize the Android malware as attack features and evasion features and present them in a meta model. Different from focusing on requirements in malware ontology analysis, we consider and maintain the traceability between the features and their corresponding code in MYSTIQUE.

- Based on the meta model, we develop an SPL architecture to generate new Android malware by an multi-objective evolutionary algorithm. Our approach is implemented in an automated framework named MYSTIQUE. In malware generation, we propose a Behavioral Description Language to support model-driven code assembly and verify constraints in code assembly.

- We survey and evaluate the state-of-the-art AMTs using our generated malware. The experiments show that the existing AMTs are quite weak at detecting these new malware — a detection ratio of less than 30% on average. We propose the countermeasures in order to detect the malware generated by MYSTIQUE.

- We have generated over 10,000 samples of Android malware by combining different attack and evasion features. They can serve as a benchmark to assess detection capabilities of AMTs, as they cover different representative combinations of features, which are missing in the current malware benchmarks.

- To evade the AMTs detection, we extend MYSTIQUE to MYSTIQUE-S that generates malicious code and sends it to the client side on the fly; the client app also executes the malicious code dynamically. To facilitate the integration with other tools for AMT auditing, we adopt a service-oriented architecture for MYSTIQUE-S.

- We conduct evaluation of MYSTIQUE-S on 16 real-world Android devices with various AMTs installed. We submit the client-end app of MYSTIQUE-S onto the GooglePlay, and it can pass the manual vetting process. We also use MYSTIQUE-S to construct the dynamic attacks for several recently popular malware cases, and the attacks are successful.

4. Large-scale Analysis of Android Malware. As the most popular operation system on mobile devices, Android is facing a serious situation against a large amount of attackers, who have released millions of Android **P**otential **H**armful **A**pplications (PHAs) to disturb users' normal life. It is highly desirable to have a comprehensive understanding of Android PHAs for the purpose of protection, detection, prediction and forensics. Therefore, we propose a systematic approach to study Android PHAs, unveil

security issues, obtain insightful conclusions and highlights, and predict the future trend for research. We have collected 4,267,178 Android apps from a variety of Android marketplaces, where 1,004,550 PHA variants are identified and analyzed. Different from previous works, this work focuses on the differences and evolution of apps' characteristics, and identifies multiple security-related issues concerned by both academia and industry. In order to provide a comprehensive view for these issues, we propose four analyses on individual app, malware family, malware author, and market, to conduct our study and guide the analysis. Furthermore, we propose six dimensions to cluster apps for different analysis tasks to achieve efficiency and accuracy in the large-scale analysis. Some of the key findings reflect the characteristics of attacks, and the weaknesses in protection, which can benefit all stakeholder.

Contributions:

- We propose a representation model, which can depict the basic information and characteristics of Android app. We leverage representation model for the large scale app and PHA analysis.
- We propose 15 features to characterize Android app or PHA. We also apply frequent-itemset mining to mine those high-order features that are the frequent value sets of basic features. Further, we can measure PHA similarity in terms of the six dimensions, e.g., time and market.
- We report the findings of our large-scale app analysis. We shed light on the authorship and auto-generation of PHA, the relationship of PHA and vulnerability, the correlation of PHA families, and the detection mechanism of markets.
- Based on the analysis model and dimensions, we have investigated 4,267,178 Android apps, including 23.5% of them as PHA inside. We publish this huge PHA data set for research purpose.

To sum up, four works are included in the thesis: in order to understand malware and its evolution, we conduct the work to extract the essence of Android malware by clone detection and differencing analysis, and build a semantic model for malware using DSA

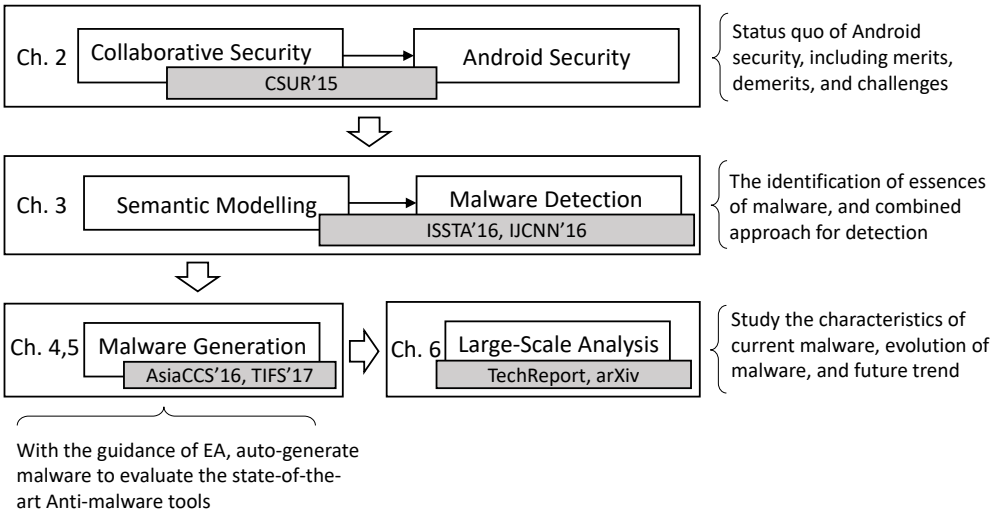

FIGURE 1.2: The roadmap of this dissertation

which cover multiple malware variants; we develop a framework SMART which combines machine learning and static analysis to detect malware; the work MYSTIQUE, which can automatically generate malware of privacy leakage, is leveraged to evaluate the performance of existing anti-malware tools. Moreover, we extend this work to MYSTIQUE-S which employs dynamic code loading to distribute malicious code. The generated malware is used to audit the capabilities of anti-malware tools detecting malware with dynamic code loading; last, we conduct a large-scale analysis of Android malware to obtain insightful conclusions and highlights. The results are beneficial and useful for the security community to understand the state affairs and future trend, thereby improve and enhance the protection of the Android ecosystem.

1.3 Thesis Outline

Fig. 1.2 shows the roadmap for this dissertation, which describes the sequence of our research work and thereby reveals the train of our thoughts on Android malware. The remaining of the thesis is organized as follows:

In Chapter 2, we present the background and related work on Android malware detection, introducing the status quo of Android security and identifying the detection mechanisms and malware datasets used in this thesis.

Chapter 3 describes the semantic modelling of Android malware, and a combined approach for malware detection and classification. In this section, we propose the essence of Android malware, and use it to better depict Android malware, subsequently improve the detection approach.

In Chapter 4, we build a feature model for attacks in malware, and employ software product line to automatically generate malware. The generated malware is used to evaluate the existing anti-malware tools.

In Chapter 5, we employ the dynamic code loading technique to distribute the generated malware, and explore the capabilities of existing anti-malware tools detecting such kinds of malware.

In Chapter 6, we perform a large-scale analysis on collected Android apps, which contains over 1 million malware. This work contains five analyses of Android ecosystem – app analyse, malware analyse, authorship analyse, market analyse, and vulnerability analyse.

In Chapter 7, we conclude the works in the thesis, and propose the potential directions in the future.

1.4 Publication List

Most of the work presented in this thesis are either published or accepted in the following international conferences and journals. As shown in Fig. 1.2, the work in Chapter 2 is mainly from the paper in *ACM Computing Surveys, vol. 48, no. 1, 2015*; the work in Chapter 3 is from the papers in *The International Symposium in Software Testing and Analysis, 2016* and *The International Joint Conference on Neural Networks, 2016*; the work in Chapter 4 is from the paper in *The ACM Asia Conference on Computer and Communications Security, 2016*; the work in Chapter 5 is mainly from the paper in *IEEE Transactions on Information Forensics and Security, 2017*, and; the work in Chapter 6 is from two technical reports.

All publications of the Ph.D candidate are listed as follows, and the publications that are not included in the thesis are marked with an asterisk:

1. **Guozhu Meng**, Yang Liu, Jie Zhang, Alexander Pokluda, and Raouf Boutaba, 2015. Collaborative Security: A Survey and Taxonomy. ACM Computing Survey 48, 1, Article 1 (July 2015), 42 pages.

2. **Guozhu Meng**, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang and Annamalai Narayanan, Semantic Modelling of Android Malware for Effective Malware Comprehension, Detection and Classification, The International Symposium in Software Testing and Analysis (ISSTA 2016) , University, Saarbrücken, Germany on July 18–20, 2016.

3. Annamalai Narayanan, **Guozhu Meng**, Yang Liu, Jinliang Liu and Lihui Chen, Contextual Weisfeiler- Lehman Graph Kernel For Malware Detection, 2016 International Joint Conference on Neural Networks (IJCNN 2016), July, Vancouver, Canada 2016.

4. **Guozhu Meng**, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang and Tieming Chen, Mystique: Evolving Android Malware for Auditing Anti-Malware Tools, 2016 ACM Asia Conference on Computer and Communications Security (AsiaCCS 2016), May, Xi'an, China, 2016.

5. Yinxing Xue, **Guozhu Meng**, Yang Liu, Tian Huat Tan, Jun Sun, and Jie Zhang. "Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique," in IEEE Transactions on Information Forensics and Security, vol. 12, no. 7, pp.1529-1544, July 2017.

6. **Guozhu Meng**, Yinxing Xue, Yuekang Li, JingKai Siow, Yang Liu, Kai Chen, and Jie Zhang, "The Vampire Diaries: A Large-Scale Analysis of Android Malware", TechReport.

7. **Guozhu Meng**, Matthew Patrick, Yinxing Xue, Yang Liu, and Jie Zhang, "Towards Modelling of Large-scale Android Malware Spread Between Markets", TechReport.

8. * Liang He, **Guozhu Meng**, Yu Gu, Cong Liu, Jun Sun, Ting Zhu, Yang Liu, Kang G. Shin, Battery-Aware Mobile Data Service, IEEE Transactions on Mobile Computing.

9. * Guangdong Bai, Jike Lei, **Guozhu Meng**, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu and Jinsong Dong. AuthScan: Automatic Extraction of Web Authentication Protocols From Implementations the 20th Annual Network and Distributed System Security Symposium (NDSS 2013), San Diego, CA United States, 24-27 February 2013

2

Background and Preliminaries

In this section, we present the background of Android, and preliminaries involved in the thesis.

2.1 Android System

Android is a mobile operating system developed by Google since 2008. Nowadays, it has gained the dominance of mobile operating system with a share of 87.6% [6]. The architecture of Android is shown in Fig. 2.1. Basically, Android consists of four layers. The bottom layer is Linux kernel, where drivers and system-level services are running. The upper level deploys system libraries and Android Dalvik virtual machine, which provides a variety of functionalities and runtime environment for Android applications.

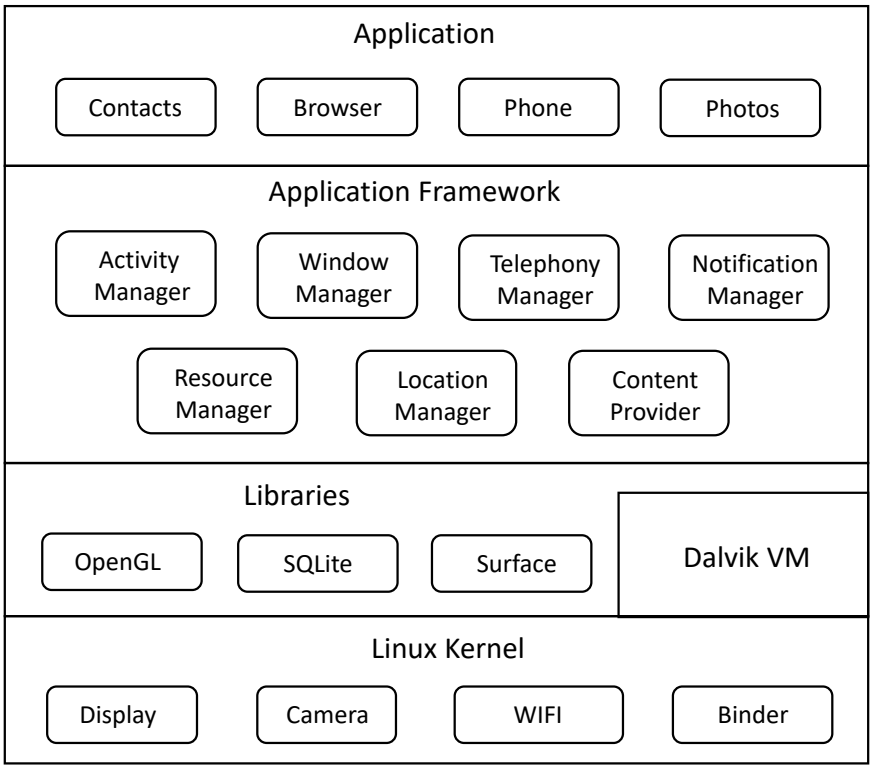

FIGURE 2.1: The architecture of Android system

There are many services written in Java in the application framework layer. For example, *activity manager* is used to manage activities in the system. The top layer is application layer, where Android applications are running. Android apps can invoke APIs or services in the framework layer and library layer to complete their functionalities.

Application Component Android apps are written in Java, and comprise of four building blocks: activity, service, broadcast receiver, and content provider. An activity is the entry point for interacting with the user. It usually has an graphical interface; a service is a component running in the background to execute long-running or expensive operations; a broadcast receiver is used to listen to the broadcast message sent either by the system or other apps, can react with predefined actions, and; a content provider is used to manage a shared set of app data that you can store in the file system.

2.2 Android Malware

We have witnessed the rapid development and evolution of Android malware since the first Trojan malware was discovered in 2010 [7, 8]. Android is facing a variety of threats that can disturb users' normal life. In this section, we identify five types of Android malware as well as their description.

Privacy leakage. Privacy leakage has received much public attention. Android malware may steal sensitive information on Android devices, such as SMS messages, contact information, geography locations and call logs [9]. The stolen information can be used to track users, make profits, obtain Mobile Transaction Authentication Number (mTAN) and so on. Privacy leakage constitutes a large portion of Android malware (about 78.7% in GENOME).

Privilege escalation. It is common to grant privileges to an application upon installment. However, vulnerabilities in these applications can result in an increase of privilege authorizations. Permissions on Android, for example, must be explicitly identified and applications cannot access the device's resources until the installer grants it the required permissions. However, many malicious applications circumvent the permission mechanism and exploit indirect tactics to access sensitive resources. As Grace *et al.* discuss in [10], permission mechanisms can be infiltrated by malicious applications calling other applications which have their authorized permissions; *RageAgainstTheCage*, *Exploid* and *Zimperlich* are three sorts of typical exploits of Android vulnerabilities which are employed to elevate the privilege of applications [11, 12].

Financial charge. Premium Rate Services (PRS) are value-added services provided by a telecom provider. Examples of PRS include subscriptions to information, services of gaming, charity donations and so forth, which charge users beyond the standard network charges. Android malware can automatically and stealthily text or call a premium number, and cause additional fees [13].

Phishing. This attack uses social engineering techniques and disguises malware to be

a normal app, which tricks users into exposing their credentials. Phishing attack is becoming progressively severe, especially after it targets the financial apps [14]. SmiShing, a kind of phishing attacks, spreads fake SMS messages to users and tricks them into opening the crafted phishing web page and entering their credentials. In addition, malware can also mimic target apps (e.g., banking apps and social apps) installed on device. The credentials entered by users in the fake app will be sent to the attacker.

Extortion. Since the first ransomware Simplocker was discovered in 2014 [15], a large amount of variants have swarmed into mobile devices in 2015. The attack of extortion in ransomware basically contains two subsequential behaviors—encrypting the files in the accessible storage via cipher, and deleting the original files. After receiving the ransom from the user, the attacker may (or may not) release the encryption key for the victims to decrypt the files.

Summary: It is observed that malware variants often share similar code, especially for variants of the same attack. As reported by Crussell *et al.* [16], software clones are common. Recently, Chen *et al.* [17] detect malware based on the clone detection techniques. Thus, code clone analysis helps to identify common malicious code among variants which introduces our work in Chapter 3. The characteristics of modularity of Android malware allows us to conduct the works in Chapter 4 and Chapter 5, in which we build feature model for attack behaviors and use auto-generation techniques to automatically generate Android malware.

2.3 Android Defense

There are dozens of defense techniques against Android malware [18, 19]. In this section, we brief existing anti-malware techniques, and explore the detection mechanisms employed by these tools.

2.3.1 Brief on Anti-Malware Techniques

Machine Learning. Known malware is fingerprinted, and represented with a number of features. The features can be used to identify similar Android malware and its variants. It first extracts specific features to represent Android malware, and perform a similarity computation in terms of these features to identify Android malware. For example, DREBIN [20] extracts eight types of features to represent Android malware, MUDFLOW [21] takes the link from source to sink as feature to identify Android malware.

Static Analysis. Static analysis inspects Android apps without executing them. Generally, it achieves better code coverage and scalability comparing with dynamic analysis, as it can get the source code or recovered code easily. However, it is susceptible of obfuscation techniques and dynamic code loading techniques, which make static analysis inaccurate. There are many static analysis approaches such as FLOWDROID [22], ICCTA [23], MUDFLOW [21], and APPCONTEXT [24], which can identify the information flow for sensitive data in Android.

Dynamic Analysis. Different from static analysis, dynamic analysis needs to execute Android apps to get evidence with regard to the analysis target. Android apps are installed in an emulator or a real mobile device, and execute with a certain crafted input. Dynamic analysis either collects the execute traces or tracks the flow of sensitive information, and then determine whether the analyzed app is malicious or not. TAINTDROID [25], MOCKDROID [26], and BAYESDROID [27] are typical approaches of dynamic analysis. Dynamic analysis overcomes the shortcomings of static analysis, but is limited to scalability and code coverage.

Security testing. Security testing is a process of exploring as many as functionalities of software to determine whether it has any flaw or malicious behaviors. Generally, security testing is a combined technique with dynamic execution, instrumentation, and even static program analysis. For example, [28, 29] use testing to detection race conditions in Android apps, [30, 31] use fuzzing testing to explore the potential vulnerabilities in

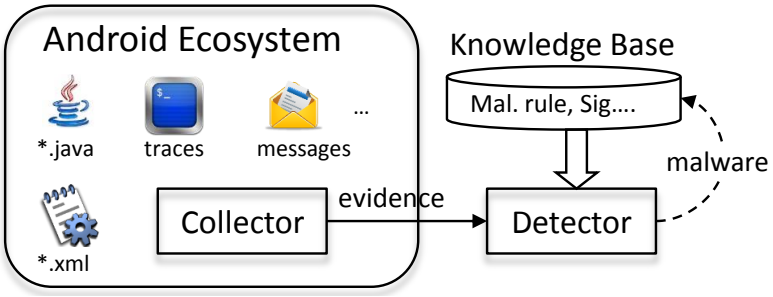

FIGURE 2.2: The working process of an AMT

Android apps or Android system. In addition, security testing is also used in detecting GUI bugs in Android apps [32–34].

Anti-virus tools. Many companies In order to accelerate the detection and reduce false positives, many anti-virus software compute hash value for Android malware as signature, and identify malware with signatures.

2.3.2 Detection Mechanism

In this section, we explore the capabilities of off-the-shelf AMTs. Fig. 2.2 shows the working process of a typical AMT. Basically, it contains *Collector* which collects evidence (§ 2.3.2.1) from Android ecosystem. The evidence is sent to a *Detector* which determines whether the app is malicious or not. Generally, the detector makes the decision based on *Knowledge Base* (§ 2.3.2.2). Additionally (as shown by dashed line), the confirmed malware can be supplemented into the knowledge base to update the knowledge base. Finally, according to our survey and testing on AMTs, we list detection mechanisms adopted by mainstream AMTs in the next section.

2.3.2.1 Evidence Collection

AMTs need to collect and extract evidence as proof to identify Android malware. Evidence can be collected from various sources in Android ecosystem, ranging from different program entities (e.g., *AndroidManifest.xml* and class files) to various program output of dynamic execution. We give a list of evidence as follows.

- **Used permission (UP).** Android malware need to acquire specific permission provided by Android to execute some risky operations. This evidence is collected by works [20, 35–38].

- **Hardware components (HC).** *AndroidManifest.xml* lists the hardware components that are used in the app, e.g., an app needs to include feature `android.hardware.camera` to access the camera [20].

- **Android components (AC).** There are four types of building blocks of Android, i.e., *activity, service, content provider and broadcast receiver*. The manifest file keeps a list of these components, the declaration in which is used to identify malware [20].

- **Filtered Intents (FI).** Different components can communicate with each other via *Inter-Component Communication (ICC)* [23, 39]. The intent filters defined in the manifest can control the components that own an ICC channel [20].

- **Android APIs (AA).** Android provides abundant APIs to support the development. However, APIs might be abused for malicious intents, e.g., the API `sendTextMessage` can be used to send spam SMS. These APIs are hints for malware in [20, 21, 38, 40, 41].

- **Constant value (CV).** There exist many constant strings in Android which define the semantics of specific code. For example, [20, 24] collect the network addresses used by apps, which could be the address of a C&C server.

- **Control flow (CF).** Control flow is either explicit or implicit. The explicit control flow can use direct call of methods [37, 40, 42, 43] to execute malicious behaviors, while implicit control flow relies on the special dispatch mechanisms in Android.

- **Data flow (DF).** Data flow analysis is to track the flow of data in Android. It can be achieved by tainting memory, file of Android [25], Dalvik heap [40], static analyzing Android code [42], and the information links between different Android APIs [41]. Data flow characterises the malware, especially privacy leakage.

- **Program dependency graph (PDG).** PDG defines the dependency relationship between statements or methods in Android, which captures essential program semantics. Works [16, 44] extract PDGs from apps as evidence to identify malware.
- **Execution trace (ET).** The execution traces of Android apps are strong evidence for identifying malware. In [45], patterns of the system calls appearing in malware are collected for detection. As malware often launches attacks via HTTP or SMS, [46] collects the HTTP behaviors to fingerprint Android apps.

As the attack of privacy leakage involves both data and operations, some attack hints can be found in all the above types of evidence. Other types of attack are more closely relevant to some certain types of evidence, e.g., the attack of functionality abuse is mainly behavior-oriented and highly identifiable based on Android APIs and Android Components.

2.3.2.2 Knowledge-base Detection

The knowledge base is the basis and criteria of distinguishing malware from benignware. After obtaining the evidence, AMTs that use the knowledge base confirm malware by the following strategies:

Signature of Known Malware (SKM). An abundance of malware collections [5, 20, 47] is publicly available for industrial and academic research. Such abundance of malware enables to fast detect malware based on signatures or features. For example, DROIDSIFT extracts *behavior graph* from known malware. Many commercial antivirus tools use the hash value of malware as the signature [48]. However, approaches based on exact matching with known malware (e.g., comparing hash value) are not resistent to new variants.

Attack Pattern (AP). To overcome the limitation of signature based detection, existing AMTs can leverage the knowledge of attacks, including attack targets, attack techniques, attack camouflages and so forth. We herein call it *attack pattern* [25, 49–52] — generally, some form of modeling of attack behaviors with aforementioned evidence.

TABLE 2.1: AMTs and their considerations in malware detection

Tools		Evidence										Knowledge Base	
		UP	HC	AC	FI	AA	CV	CF	DF	PD	ET	SKM	AP
Machine Learning	DREBIN [20]	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✗
	DROIDAPIMINER [53]	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✗
	ADAGIO [54]	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗
	DROIDSIFT [44]	✗	✗	✓	✗	✓	✓	✗	✗	✓	✗	✓	✗
	ALLIX *et al.* [43]	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✓
	REVEALDROID [41]	✗	✗	✓	✓	✓	✗	✓	✓	✗	✗	✓	✗
Static Analysis	SCANDROID [55]	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
	APPOSCOPY [42]	✗	✗	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓
	FLOWDROID [22]	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
	DROIDSAFE [40]	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
	MASSVET [56]	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗
	ICCTA [23]	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
Dynamic Analysis	TAINTDROID [25]	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✗	✓
	CROWDROID [45]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗
	COPPERDROID [57]	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓
	DROIDSCRIBE [58]	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓
Security Testing	DROIDFUZZER [31]	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓
	SWIFT-HAND [32]	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓
	GUITAR [33]	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓
	Bielik *et al.* [28]	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓
	RACERDROID [29]	✗	✗	✓	✗	✓	✓	✗	✗	✗	✓	✗	✓
	TRIMDROID [34]	✗	✗	✓	✗	✗	✓	✓	✗	✓	✓	✗	✓
Hybrid Approach	APPCONTEXT [24]	✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗
	MUDFLOW [21]	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✓	✗
	SMART [59]	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗
Anti-virus Tools	VIRUSTOTAL [60]	?	?	?	?	?	?	?	?	?	?	?	?

2.3.3 Summary

It is non-trivial to understand the status quo of malware detection and employed detection mechanisms. In this section, we investigate many state-of-the-art AMTs and explore the hidden detection mechanisms. Table 2.1 shows state-of-the-art AMTs of six kinds of malware detection techniques as well as their defense mechanisms. This investigation inspires us to conduct a work (see Chapter 3) to detect malware which combines static analysis and machine learning to achieve scalability and effectiveness.

In addition, the work MYSTIQUE in Chapter 4 selects 9 state-of-the-art AMTs in Table 2.1 and 57 anti virus software as evaluation subject, and generates 10,000 malware to evaluate these tools. This work evaluates the weakness and shortcomings of AMTs against Android malware on the one hand, and on the other hand, it evaluates the features (a.k.a. evidence) of Android malware and identifies the optimal combination of these features that can easily bypass the detection but contains as many as attack behaviors. Similarly, the work MYSTIQUE-S in Chapter 5 integrates dynamic code loading into generated malware, and evaluate the performance of AMTs detecting such kind of malware. The work in Chapter 6 refers to the features proposed in this section, and use many of them to depict the characteristics of Android malware and propose many insights from machine learning and statistics inference.

2.4 Android Malware Dataset

To evaluate our works in this thesis, we have been collecting a tremendous amount of apps (including malware) from Sep. 2013 to Jun. 2016. The apps originate from 32 different sources (Table 2.3 shows the detailed information of these sources, for example, the region, the language and the number of collected apps). These sources can be categorized into two types—malware benchmarks and Android markets, which are described as follows:

Android Application Repositories. In order to study emergence, characteristics, and evolution of Android malware, we have crawled 4,267,178 Android apps from different Android application repositories including GOOGLEPLAY and ANDROIDZOO. Among all the apps we collect, 1,004,550 samples are labelled as malware by VIRUSTOTAL and the remaining are not or unrecognized. Before the analysis, we compute the hash value of apps as signature (here we use sha256 checksum) to identify the distinct app. Totally, there are 4,039,468 (94.9%) distinct apps in this dataset, which means that there have replicas of apps across different marketplaces (An app in a market is replica if it can be found in another market). Among all these marketplaces, apps from GOOGLEPLAY accounts for 21% of all the collected apps. Most of other wild unlabelled apps come

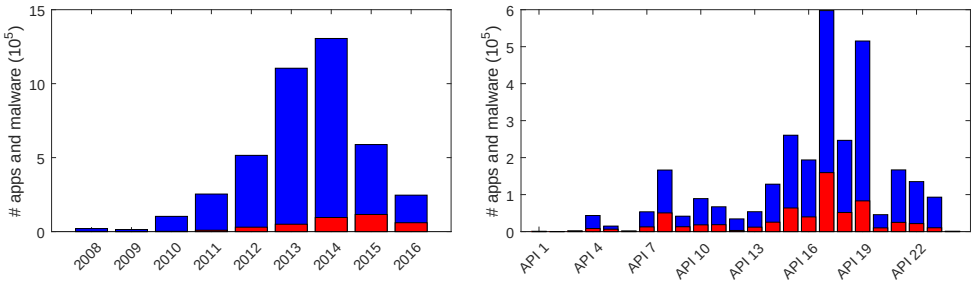

FIGURE 2.3: The statistics of apps and included PHA. The x-axis of the left figure is the years since the PHA is created, and the y-axis denotes the number of apps and PHA. The x-axis of the right figure is the Android versions from API 1 to API 24, and the y-axis denotes the number of apps and PHA. The blue bar denotes the number of apps, and the red bar denotes the number of PHA.

TABLE 2.2: The percentage of replicas in markets. We list top 5 markets with the maximal percentage and top 5 with the minimal percentage.

	fdroid	freewarelovers	apkpure	apk20	coolapk
Ratio	92.4%	67.1%	65.3%	51.3%	51.0%
	apkmirror	getjar	appfun	hiapk	chinamobile
Ratio	13.3%	13.0%	9.8%	8.7%	7.9%

from the marketplaces that are hosted in China and Europe, etc. As shown in Table 2.2, the market APPSAPK has the minimal proportion (7.9%) of replicas while FDROID has the maximal proportion (92.4%). We use apps and malware in this dataset as analysis subject, and study the characteristics, evolutions, and detection mechanisms of Android malware in Chapter 6.

Fig. 2.3 shows the distribution of apps and PHA in two different dimensions. In particular, the first figure shows the number of apps and PHA along with the date of year, the second figure shows the number of apps and PHA targeting different Android OS versions.

Malware benchmark. We have collected three well known malware datasets: GENOME [5], DREBIN [20], and VIRUSSHARE [47]. As shown in Table 2.3, they contains thousands of Android malware which are already confirmed for academic and industrial use.

In addition, we take into account the labelled PHA reported by VIRUSTOTAL. We submit all unlabelled apps to VIRUSTOTAL for PHA detection. If any antivirus product

from VIRUSTOTAL reports the submitted app as PHA, we consider this app as PHA. If the app is detected as PHA, the antivirus product can give the family name which groups a set of PHA variants with similar malicious behaviors. However, the family names vary from different companies. Therefore, we employ AVCLASS [61] to normalize family names, and obtain 1,400 malware families containing 200,373 (20.0%) malware samples in total.

The malware in GENOME and DREBIN are well-labelled, and malware samples with similar malicious behaviors are clustered into the same family. Based on that, we conduct the work in Chapter 3 to extract essences from malware samples in the same family, and propose the approach to detect and classify malware. The essence composes the meta model in Chapter 4 and 5, which can be used to guide the auto generation of Android malware.

TABLE 2.3: The description for all sources. It contains three PHA datasets in the table—GENOME, DREBIN and VIRUSSHARE.

Name	URL	Language	Country	# apps	# PHA
androiddrawer	http://www.androiddrawer.com/	English	Global	9,106	747
androzoo	https://androzoo.uni.lu/	English	Global	2,727,911	612,584
anruan	http://www.anruan.com/	Chinese	China	6,781	3,233
anzhi	http://anzhi.com/	Chinese	China	113,041	59,570
apk20	http://www.apk20.com/	English	US	33,555	5,260
apkmirror	http://www.apkmirror.com/	English	US	11,641	404
apkpure	https://apkpure.com	English	US	2,827	435
appchina	http://www.appchina.com/	Chinese	China	14,769	8,858
appsapk	http://appsapk.com	English	US	1,997	196
baidu	http://shouji.baidu.com/	Chinese	China	9,606	425
chinamobile	http://mm.10086.cn/	Chinese	China	236	131
cnmo	http://app.cnmo.com/	Chinese	China	10,745	3,831
coolapk	http://coolapk.com	Chinese	China	13,851	2,754
drebin	https://www.sec.cs.tu-bs.de/~danarp/drebin/	Global	Global	5,560	5,560
eoemarket	http://www.eoemarket.com/	Chinese	China	27,012	12,257
fdroid	https://f-droid.org/	English	US	3,757	64
flyme	http://app.flyme.cn/	Chinese	China	11,278	4,512
freewarelovers	http://www.freewarelovers.com/android	English/German	Germany	3,595	248
genome	http://www.malgenomeproject.org/	Global	Global	1,260	1,260
getjar	http://www.getjar.com/	English	Europe	53,867	22,535
gfan	http://apk.gfan.com/	Chinese	China	10,746	5,533
googleplay	https://play.google.com/store?hl=en	English	Global	894,290	84,779
hiapk	http://www.hiapk.com/	Chinese	China	29,291	11,555
huawei	http://appstore.huawei.com/	Chinese	China	1,771	0
mob	http://mob.org/	English	US	2,914	567
mumayi	http://www.mumayi.com/	Chinese	China	45,399	21,219
qq	http://sj.qq.com/myapp/	Chinese	China	144,944	85,546
slideme	http://slideme.org	English	US	4,711	209
virusshare	https://virusshare.com/	Global	Global	24,322	24,322
wandoujia	http://www.wandoujia.com/apps	Chinese	China	2,888	1,641
wangyi	http://m.163.com/android/index.html	Chinese	China	7,054	3,447
xiaomi	http://app.mi.com/	Chinese	China	36,453	20,868
Total				**4,267,178**	**1,004,550**

3

Semantic Modelling of Android Malware for Malware Detection

3.1 Introduction

Android has become a popular mobile operating system installed on over 1 billion devices, which accounts for more than 81.5% of all smartphones in 2014 [2]. Its prevalence and the prosperity of Android marketplaces have made it a hot target for attackers to upload various malware. The total number of mobile malware samples has increased by 112%, and exceeds 5 million [62] in 2014.

Existing approaches to detecting Android malware typically rely on bytecode similarity analysis [16, 63, 64], machine learning techniques [20, 44, 45, 53, 65–68] or information flow analysis [10, 21, 22, 25, 42, 69]. Bytecode similarity analysis usually adopts pair-wise comparison, with time complexity $O(n^2)$ for comparing n possible malware variants. Machine learning based approaches can be effective and efficient in detecting malware, but only provide some predictive features for malware without explaining the malicious behaviors involved. The approaches relying on information flow analysis are accurate, but not efficient for the large-scale detection.

The key in successful defense to malware variants or zero-day attacks is to understand the different attack behaviors in malware and prevent them in a proactive way. However, none of the aforementioned approaches provides a complete semantic explanation of attacks, the behaviors involved in attacks, and the actions inside behaviors.

In this thesis, we propose an explicit semantic model to capture the malicious behaviors of a malware family so as to achieve a precise comprehension. We represent the essential elements of malicious behaviors in malware variants as Deterministic Symbolic Automata (DSA) [70] with transitions being system APIs (see Fig. 3.1). The motivation is that one malicious behavior may occur in a variety of manners, which can easily bypass malware detection tools based on malware signatures or patterns. For example, there are multiple ways of sending private data from mobile devices to a remote server, e.g., using an `HttpURLConnection` object to write a stream to the server, or launching a browser to attach the data in the URL. Although the two implementations differ notably, they fulfil the same task. DSA can encode alternative transitions among states, which enables capturing malware variants of the same kind.

The ultimate goal of our study is to build a semantic model of malware towards effective detection, classification, and forensics. To the best of our knowledge, it is the *first attempt* to propose a semantic model considering malware variants.

Based on the semantic model, we implement an automatic malware analysis framework, named **S**emantic **M**odelling of **A**nd**R**oid a**T**tack (SMART), which provides three key features—automatic semantic model construction from existing malware families, efficient malware detection and malware classification. The overall workflow of SMART

is shown in Fig. 3.3. First, SMART starts with identifying semantic clones [71] among malware variants to capture the common malicious behaviors. We propose a bytecode clone detection method (§ 3.5.1) to effectively identify similar malicious code in the form of bytecode clones. To ensure the scalability and eliminate the noise of third party libraries, SMART filters out commonly-known libraries according to [72]. For each clone set, we introduce an efficient bytecode differencing algorithm to build a DSA by comparing n clone instances at one time with the complexity $O(n)$ (§ 3.5.2). From DSA, low level actions are extracted (§ 3.5.3). Finally, a semantic model in Fig. 3.1 is stored into our DSA database. DSA is able to cover most of the attack, but not for all. The applicability and generality of the DSA model is discussed in Section 3.8.

Once the semantic models are constructed, we extract malicious features from DSA and perform malware detection via machine learning techniques (§ 3.6.1). Then, for each suspicious app, SMART performs a static analysis to check whether malicious behaviors from our DSA database is contained in the app via a DSA inclusion check (§ 3.6.2). If DSA inclusion succeeds, we can confirm the maliciousness of the app and malware families they belong to. Otherwise, we identify the actions that are contained in the app for the sake of understanding its behaviors. The DSA-based semantic model facilitates finding new malware variants, new families or even new attacks. The machine learning and the static analysis complement each other to achieve both good performance and high accuracy. In summary, we make the following contributions:

- We propose a hierarchical semantic model of Android malware based on DSA, which links malware family, malicious behaviors and fine-grained actions. To the best of our knowledge, this is the first work in this direction.
- We propose an effective method to extract the common malicious behaviors as DSA from malware variants of an Android malware family.
- We combine machine learning and static analysis to detect and classify malware. The first phase quickly identifies suspicious apps, and the second phase uses DSA inclusion to further confirm and classify the suspicious apps. Hence, we can achieve both scalability and accuracy.

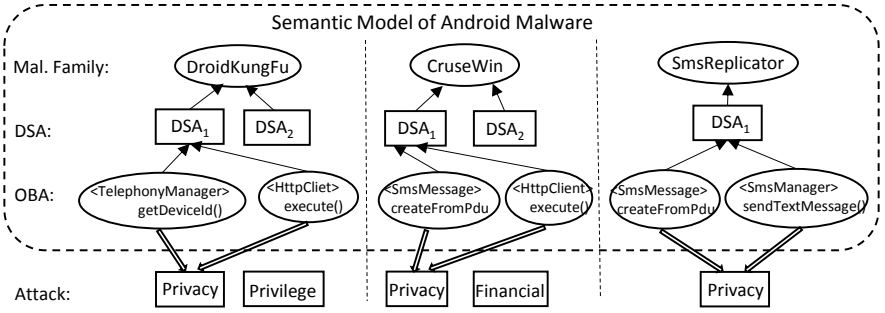

FIGURE 3.1: The hierarchy of the semantic model

- We test SMART on 5,560 known malware and 223,170 real-world apps. The results show that SMART outperforms many anti-virus (AV) tools and academic approaches, with a precision of 86.7% for malware detection. For wild prediction, it detects 4,583 new malware variants.

3.2 Related Work

There are substantial number of related work on android malware analysis and also a comprehensive survey [4]. Here we list some most relevant work as follows.

Bytecode similarity analysis. As malicious repackaged apps usually share the common basic functions and features, the idea of DROIDMOSS [64] relies on the pair-wise check of the similarity between two apps and uses it as the basis to detect repackaged apps. DROIDMOSS proposes to generate fingerprint via a fuzzy hashing technique to localize and identify the code changes due to repackaging behaviors. Similarly, DNADROID [16] pair-wisely measures the similarity between two apps to find the cloned attacks. Similarity score is calculated by applying the VF2 algorithm [73] to compute *subgraph isomorphisms* existing in program dependency graphs (PDGs) between methods in candidate apps. To overcome the scalability limitation due to pair-wise comparison, PIGGYAPP [63] and WUKONG [74] adopt the clustering technique to address the piggybacking and clone detection among similar mobile apps.

Mining or learning malicious behaviors. A precise model of malicious behaviors can significantly improve the accuracy of detection. Previous efforts have been made to

distinguish malicious apps from benign ones by classification. CROWDROID [45] uses Android API call sequences in Linux kernel as features. DREBIN [20] adopts static analysis to extract features relevant to malicious attacks, e.g., permissions and API calls. DROIDAPIMINER [53] extracts features from dangerous Android APIs calls, APIs parameters and package level information for the classification. MAST [75] selects permissions, Intent filters, the existence of native code and zip files, then applies *Multiple Correspondence Analysis* on GENOME malware collection. Peiravian *et. al.* [38] take permissions and API calls as features to train a model to detect malware. DROIDMINER [65] proposes a two-tiered behavior graph to model malicious program logic into a vector of threat modalities, and then applies classification according to these modalities. DROIDSIFT [44] further models API-relevant behaviors into weighted contextual API dependency graphs and classifies malware based on these graphs. Recently, APPCONTEXT [24] proposes to differentiating malicious and benign behaviors based on the contexts: *events* and *conditions* that trigger security-sensitive behaviors. Predictive features are further extracted from the differentiating results.

There also exist some works on malware model using automata. Babić *et al.* [76] use *tree automata* to model malicious behaviors in malware. They use the extracted system call dataflow dependency graphs to infer k-testable tree automata. Preda *et al.* [77] propose *abstract symbolic automata (ASA)* to present the syntactic and semantics of binary executables. Aiming at identifying the messages between malware and the environment, Bonfante *et al.* [78] conduct a dynamic analysis on malware, and employ automata to discriminate types of different execution by an arbitrary message. SMART is the first attempt to use automata to describe a whole malware family, with describing the common and different parts across the malware variants.

Malware detection with other techniques. The traditional malware detection is based on signature or hashing matching [79, 80], which is incapable of addressing the obfuscation and malware variants. Information flow analysis is another effective approach in malware detection, which usually adopts dynamic or static taint analysis for tracking information-flow in mobile apps. For instance, TAINTDROID [25] and VETDROID [81] use dynamic taint analysis by instrumenting the Dalvik VM, while APPOSCOPY [42] and [10, 22] employ static taint analysis for scalability. In APPOSCOPY, results of static

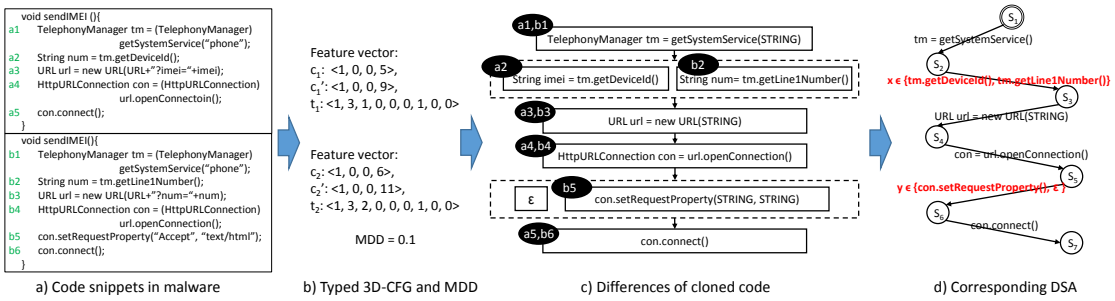

FIGURE 3.2: A running example of representing malicious behavior in DSA for two samples from DroidKungFu

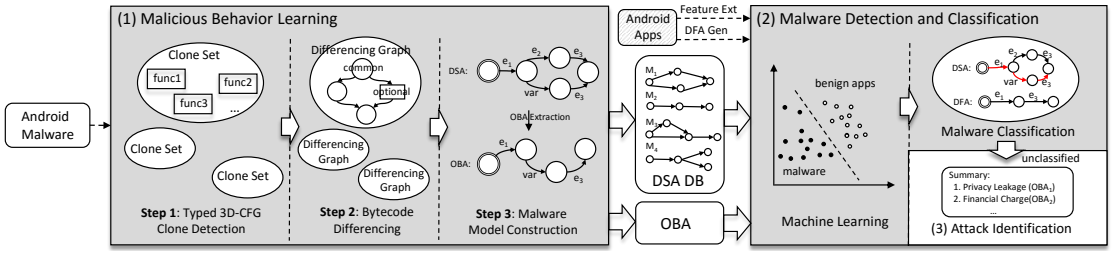

FIGURE 3.3: The overview of our approach

taint analysis are combined with high level signatures of malware to further speed up the detection. Recently, MUDFLOW [21] leverages static taint analysis to detect information leaks via comparing source-to-sink patterns in malware or benignware, and massively mines app repositories for patterns of *normal* data flow. The mined patterns are used as predictive features for detection. DROIDPF [82] and [83] employ the software modelling checking technique to verify Android apps and identify the potentials vulnerabilities and bugs existing in Android apps.

Different from the previous works, we emphasise the malware model considering the possible variants. Our proposed semantic model, which captures commonality and variety of malicious behaviors, helps to understand the essence of the attack behind. Our combined approach of malware detection has advances in resisting both obfuscation and variants of malware, and overcomes the challenges in scalability.

3.3 Modelling Android Malware

In the program comprehension domain, *deterministic finite automaton (DFA)* has been used to model the program logic of a method in Android app [84, 85].

DFA can depict a concrete malicious behavior exactly, but it is inflexible in presenting a behavior with many variants. Thus, inspired by the fuzzy Android code search engine CODOTA [86], we use DSA [70] to model the program logic of one malicious behavior with variants.

Definition 1. A deterministic symbolic automaton (DSA) is a 6-tuple $(\mathscr{Q}, \Sigma, \delta, q_0, \mathscr{F}, Vars)$ where $\mathscr{Q}$ is a finite set of states; Σ is a finite alphabet, indicating executed statements in a malicious behavior; $Vars$ is another finite alphabet, and it is defined as a set of variables which can be assigned with arbitrary statements. In addition, the symbol $\varepsilon \in Vars$ denotes a null statement; δ is hereby a revised transition function: $\mathscr{Q} \times (\Sigma \cup Vars) \rightarrow \mathscr{Q}$, representing a transition relation; and $\mathscr{F} \subseteq \mathscr{Q}$ is a set of final states.

Compared with DFA, DSA is equipped with the capability of expressing variants via symbolic variables. $Vars$ in a DSA is a unique feature that covers the variety of transitions amongst nodes in the DSA. Similarly, $traces(DSA)$ is the set of all execution paths $\langle s_0, s_1, s_2, ..., s_n \rangle$ contained in the DSA, where $s_0 = q_0$, $s_i \xrightarrow{e_i} s_{i+1}$, $e_i \in \Sigma$ and $s_n \in \mathscr{F}$. Malicious apps in the same malware family may have slightly different behaviors, resulting in a set of code clones that contains one clone instance from each malware variant in the family. Therefore, each clone set can be summarized as one DSA, with $\mathscr{Q} \times \Sigma \rightarrow \mathscr{Q}$ denoting the common parts and $\mathscr{Q} \times Vars \rightarrow \mathscr{Q}$ denoting differences among clone instances.

For a malware family with many variants, we can model the malicious behaviors using a number of DSA, where each DSA is summarized from a set of cloned methods in these variants (§ 3.5). These DSA together can model the malicious behaviors shared by these variants, i.e., the signature to fingerprint this malware family. However, one DSA describes a behavior involving a series of operations on multiple targets or resources. To have a fine-grained model of the operations at implementation level, we propose the concept of Object-based Actions (OBAs) (§ 3.5.3.2) to describe low-level atomic actions relevant to a certain resource in Android OS context.

Based on DSA and OBA, we propose a hierarchical model with three layers: 1) the top layer is the malware family model, which summarizes the common behaviors of malware families. It is modelled by a set of DSA. 2) the middle layer is an abstract model

of a type of malicious behaviors along with its possible variants. DSA is the representation of malicious behaviors. 3) the bottom layer models the concrete objected-based actions (OBAs)—a partial DSA relevant to a target or resource object for composing a malicious behavior, e.g., deleting a system file or register entry.

Fig. 3.1 shows the hierarchical relationship of malware family, DSA, OBA and type of attacks. For instance, DroidKungFu [87] contains two types of malicious behaviors (DSA): privacy leakage and privilege escalation. The privacy leakage attack involves two OBAs: operations relevant to class TelephonyManager and HttpClient. The families CruseWin and SmsReplicator also contain the issue of privacy leakage, while involving different actions. Note that different OBAs can compose a specific type of attack, as the object or the resource under the OBAs decides the nature of the attack.

In our semantic model, we ignore the connections among DSA (or OBAs), which are data flow or control flow, in the part of malware family (or attack type). As the implementations of connections can vary greatly (e.g., data flow among DSA may use inter-procedural or external channels), we omit connections among DSA (or OBAs) for scalability issues.

Running Example. We select two samples from malware family DroidKungFu to show a concrete semantic model and how it is learned. One malicious behavior of DroidKungFu is to steal sensitive information (e.g., IMEI code and phone number), and send the information to a remote server. Fig. 3.2 shows how the malicious behavior is expressed in three representations: typed 3D-CFG used for fast clone detection, code statements with summarized differences, and finally a DSA used as its behavior model.

In Fig. 3.2(a), the common malicious behavior consists of two main steps: 1) obtain the private data (e.g., the IMEI code or phone number) — call the instance of TelephonyManager and invoke its method to access telephone data; 2) leak the data — create an instance of HttpURLConnection and send the data via this instance. Despite sharing the same malicious intent, two samples in Fig. 3.2(a) are different in implementation: 1) the first one fetches the IMEI code via getDeviceId, and the second one gets the phone number via getLine1Number; 2) the second one additionally invokes setRequestProperty to set up parameters for the connection.

Instead of modelling two different variants with two separate DFA or one combined DFA, we represent the two variants of malicious behavior of stealing information as a DSA, shown in Fig. 3.2(d). The *init* state of the constructed DSA has a transition which invokes the method getSystemService. The DSA proceeds in a subsequent transition which supports a variable $x \in \{\text{getDeviceId}, \text{getLine1Number}\}$. Then, the state proceeds with another variable $y \in \{\varepsilon, \text{setRequestProperty}\}$, where ε denotes a null event for the transition. Thus, y denotes an optional operation. By virtue of variables x and y, this DSA precisely models four different implementations of behaviors (i.e., four valid variants). Thus, this DSA can be used by SMART for detection of variants of malicious behaviors and malware family classification. Furthermore, by analyzing OBAs in malware, we can identify the potential attacks that a suspicious app launches. For example, a contained OBA (e.g., relevant to TelephonyManager) can be used to identify a potential information leakage — the class TelephonyManager contains many methods (e.g., getDeviceId and getLine1Number) that allow attackers to stealthily obtain sensitive phone data.

3.4 The SMART Framework

SMART performs the modelling of malicious behaviors of existing Android malware, and uses these abstract semantic models to detect unknown malware or new variants. In the modelling part, the input is the bytecode of malware variants, and the output is the constructed DSA. In the detection part, the input is the DSA and a suspicious app, and the output is the detection and classification result. Fig. 3.3 shows the system architecture of SMART, which relies on static analysis of bytecode and proceeds in two phases:

- **Malicious behavior learning.** This step is based on bytecode clone detection and code differencing analysis. First, we perform bytecode clone detection to identify security-related clones from the malware variants in a malware family. Our bytecode clone detection is designed to tolerate slight differences among clone instances. Second, we adopt a code differencing analysis to learn both the

common and optional parts of one type of malicious behaviors. Then we represent them using a unified statement graphs with summarized differences in dashed rectangles, as shown in Fig. 3.2(c). Last, we generate DSA from differencing graph and extract OBAs from DSA based on consecutive transitions of certain objects.

- **Malware detection and classification.** We propose a combined approach for malware detection to achieve efficiency and accuracy, simultaneously. First, we extract predictive features of attacks from the summarized DSA. With these features, we can identify suspicious apps via ML classification. Second, we perform a static analysis process, called DSA inclusion check, to determine if the concrete behavior model of an Android app (in the form of DFA) is semantically included in any learned DSA. A match between the DFA of the suspicious app and any DSA suggests that this app has the malicious behavior. If this app matches the majority of DSA of a certain malware family, SMART suggests this app is a variant of that family. If the DFA of this app fails to match any DSA, we extract the OBAs from its DFA, and compare them with those in existing DSA in order to identify the similar attack at a fine-grained level.

3.5 Learning Malicious Behaviors

3.5.1 Bytecode Clone Detection

3D-CFG [80, 88] is a structural feature, and used to measure the similarity between methods in/among Android apps. The basic idea in [80] is to assign a structural 3D-coordinate value for each node in the control flow graph (CFG) of a method, then calculate the mass centroid of these coordinates as the signature of the method. By checking the distance between two methods' centroids, it is decided whether they are clones.

Definition 2. (3D-CFG Representation of Method) For a given method, each node in the CFG is a basic code block. Each node has unique coordinates, denoted as a 3-tuple (x, y, z) [80], where x is its sequence number in the CFG; y is the number of its outgoing edges (control flow in CFG), and z is the loop depth of the node.

3D-CFG with type information. We propose a type-enriched 3D-CFG, which takes into account the type information of statements as a part of method signature to measure the similarity of two clone methods.

Definition 3. Given a method m, the method signature $\overrightarrow{m} = (\overrightarrow{c_m}, \overrightarrow{c_m}', \overrightarrow{t_m})$ is a feature vector containing the structural and type information of the method, where $\overrightarrow{c_m}$ is the centroid, $\overrightarrow{c_m}'$ is the weighted centroid, and $\overrightarrow{t_m}$ is a vector in which t_i depicts the number of occurrences of i-th type of statements in this method.

The centroid $\overrightarrow{c_m}$ of a method m is a 4-tuple (c_x, c_y, c_z, w):

- $w = \sum_{e(p,q) \in 3D-CFG} (w_p + w_q)$,
- $c_i = \frac{\sum_{e(p,q) \in 3D-CFG} (w_p i_p + w_q i_q)}{w}, where\ i \in \{x, y, z\}$

where $e(p, q)$ is the edge between the node p and q, (x_p, y_p, z_p) is the coordinate of the node p, and w_p is the number of statements in the node.

To emphasize the importance of invocation statements, the weighted centroid is defined as $\overrightarrow{c_m}' = (c'_x, c'_y, c'_z, w')$ for method m, where $w' = w + N$, and N is the number of invoked statements; c'_x, c'_y and c'_z are recalculated according to the new weight w', respectively. For example, for the first method in Fig. 3.2(a), the centroid is $(1,0,0,5)$ and the weighted centroid is $(1,0,0,9)$ (only four invocation statements). Note that w' is 9, the sum of w and the number of invocation statements.

The difference between the centroids of two methods m_1 and m_2 is the primary condition to evaluate their similarity:

Definition 4. The Centroid Difference Degree (CDD) of two centroids $\overrightarrow{c_1} = (c_{1x}, c_{1y}, c_{1z}, w_1)$, $\overrightarrow{c_2} = (c_{2x}, c_{2y}, c_{2z}, w_2)$ (similarly for the weighted centroids) is

$$CDD(\overrightarrow{c_1}, \overrightarrow{c_2}) = max(\frac{|c_{1x} - c_{2x}|}{c_{1x} + c_{2x}}, \frac{|c_{1y} - c_{2y}|}{c_{1y} + c_{2y}}, \frac{|c_{1z} - c_{2z}|}{c_{1z} + c_{2z}}, \frac{|w_1 - w_2|}{w_1 + w_2})$$

```
1 String m1(){
2     TM tm=getSysSer();
3     String id=tm.getId();
4     return id;
5 }
```

```
1 String m2(){
2     String res=str.rep();
3     rep.rep("suffix");
4     return res;
5 }
```

FIGURE 3.4: A false positive using the 3D-CFG approach

Given $\overrightarrow{t_1}$ and $\overrightarrow{t_2}$ that represent the vector of occurrences of statements with different types in method m_1 and m_2, respectively, *Type Difference Degree (TDD)* is defined as:

$$TDD(\overrightarrow{t_1}, \overrightarrow{t_2}) = \sum(\frac{|t_{1i} - t_{2i}|}{t_{1i} + t_{2i}}) / \mid \{i \mid t_{1i} \neq 0 \mid\mid t_{2i} \neq 0\} \mid$$

Given two methods, the method-level difference degree is defined as the maximum value of their centroid distance (CDD), their weight centroid distance (weighted CDD) and also the Type Difference Degree (TDD):

Definition 5. Method Difference Degree (MDD) of two methods $\overrightarrow{m_1} = (\overrightarrow{c_1}, \overrightarrow{c_1}', \overrightarrow{t_1})$ and $\overrightarrow{m_2} = (\overrightarrow{c_2}, \overrightarrow{c_2}', \overrightarrow{t_2})$ is

$$MDD(\overrightarrow{m_1}, \overrightarrow{m_2}) = max(CDD(\overrightarrow{c_1}, \overrightarrow{c_2}), CDD(\overrightarrow{c_1}', \overrightarrow{c_2}'), TDD(\overrightarrow{t_1}, \overrightarrow{t_2}))$$

Benefits of typed 3D-CFG. For the two methods in Fig. 3.4, the CDD for their centroids and weighted centroids are both 0 — they have only one path, both represented with a single block. Thus, they have the exactly same centroid, and will be clustered into the same clone set. The 3D-CFG method will treat them as clones. In our typed 3D-CFG, MDD is $5/12$, as TDD is $5/12$. If the threshold for MDD is empirically set to a small constant (e.g., 0.1), they are not clones.

For our example in Fig. 3.2(a), the CDD between $\overrightarrow{c_1}$ and $\overrightarrow{c_2}$, i.e., the maximum distance on the elements of 4-tuple, is calculated to be $1/13$ and $CDD(\overrightarrow{c_1}', \overrightarrow{c_2}')$ to be $1/11$, as shown in Fig. 3.2(b). To calculate $TDD(\overrightarrow{t_1}, \overrightarrow{t_2})$, we measure the average distance of the statement types that appear in m_1 or m_2 (we select 9 out of 15 types of statements defined in Soot [89] in Section 3.5.2). Since type vector $\overrightarrow{t_1}$ is $\langle 1,3,1,0,0,0,1,0,0 \rangle$ and $\overrightarrow{t_2}$ is $\langle 1,3,2,0,0,0,1,0,0 \rangle$, $TDD(\overrightarrow{t_1}, \overrightarrow{t_2}) = 1/12$. Thus, MDD for the example in Fig. 3.2(b) is $1/11$.

TABLE 3.1: Elements to compare for different statements

Statement	Type	Elements to compare
$a = new\ Class()$	IDENTITY	TYPE(a), TYPE(Class)
$a = expr$	ASSIGN	TYPE(a), TYPE(expr)
$invoke\ count(a)$	INVOKE	NAME(count), TYPE(a)
$if(a > b)$	IF-ELSE	TYPE(a|b), TYPE(>)
goto a	GOTO	
return	RETURNVOID	
switch(a)	SWITCH	TYPE(a)
return a	RETURN	TYPE(a)
throw e	THROW	TYPE(e)

Pre-filtering of non-malicious clones. To assure that the clones are related to malicious behaviors, we filter out non-malicious behaviors in three ways: 1) maintain a white list of common third-party libraries not to be considered in clone detection step. It includes 89 ads libraries, 18 social sdks, and 201 development tools [72]; 42.4% of malware samples include these common libraries. Thus, this step greatly improves the soundness of malware model learning. 2) set a threshold (denoted as θ_1) to retain the clones relevant to most variants in a family, since not all variants share the same clone set (possibly containing common malicious behaviors). For example, 11 out of 14 (78.6%) variants of family Zitmo send users' incoming SMS messages to attackers via HTTP, while the others via SMS. For the clone set sending privacy via HTTP, if the ratio of the number of its clone instances to the total number of variants in the family (78.6%) exceeds a predefined threshold (e.g., θ_1 = 75%) [90], we regard it as being attack related. 3) verify the clones according to the knowledge of malicious code. For samples in GENOME, package and class name of malicious code can be identified.

3.5.2 Bytecode Differencing

After clone detection, we can get a number of clusters of similar code at method level. We summarize them by identifying the common and also different parts of these cloned methods with a bytecode differencing algorithm. The basic idea of the differencing algorithm is borrowed from the concept of progressive alignment [91] in multiple DNA sequence alignment. Instead of using pair-wise comparison with complexity of $O(n^2)$, we compare n cloned methods in $n-1$ times. Specifically, we first compare two methods with the least MDD and get results in the form of a differencing graph (e.g., Fig. 3.2(c)). Then, the other $n-2$ ones are gradually compared to the intermediate differencing graph. Finally, a differencing graph unifying the commonality and variability among cloned methods is built.

Algorithm 1 depicts the bytecode differencing step. The input is a set of CFG G of the methods in a clone set. Given a CFG g, $v(g)$ is the set of vertex of g; $children(g, n)$ denotes the successors of the node n in g. The algorithm returns a differencing graph G_r, which aggregates the input graphs with matched common nodes and summarized optional nodes. First, we sequentialize the statements of each CFG at bytecode level via a preorder traversal (implemented using BodyExtractorWalker in Soot) (line 1); then we employ *longest common subsequence (LCS)* [92] to calculate common statements shared by these cloned methods (line 2). For example in Fig. 3.2 (c), running LCS for these two methods gives $commonTokens = \{(a_1, b_1), (a_3, b_3), (a_4, b_4), (a_5, b_6)\}$. Here, the matched statements n_1, n_2,..., to n_i from different methods are denoted as the equivalent class $\{n_1, \ldots, n_i\}$. LCS at line 2 is adopted in [92] and [93] for source code differencing, while we adopt it for bytecode differencing.

In calculating LCS, a typed approach is used to determine the equivalence of two statements, We consider 9 basic types as listed in Table 3.1. Other statement types like NOP and RET are omitted as they do not carry any semantic meaning. Most types in Table 3.1 only use type information for comparison, except that the type INVOKE that needs textual matching for the name of invoked method. The rationale is that most textual information like customized variable or method names is lost after compilation and obfuscation. Only the full method names of Android APIs are kept. In Fig. 3.2

Algorithm 1: Bytecode Differencing

Input: G is a set of CFGs of cloned methods
Output: G_r is a differencing graph

```
1  list⟨list⟨Stmt⟩⟩ tokens := preorder traversal(G);
2  list⟨set⟨Stmt⟩⟩ commonTokens := LCS(tokens);
3  let v(G_r) ← ∅ be the set of vertex in G_r;
4  for set⟨Stmt⟩ node ∈ commonTokens do
5      v(G_r) = v(G_r) ∪ node;
6  for g ∈ G do
7      for src ∈ v(g) do
8          List⟨Node⟩ dstList := children(g, src);
9          Node nSrc = findNode(G_r, src);
10         for dst ∈ dstList do
11             Node nDst = findNode(G_r, dst);
12             G_r.addEdge(nSrc, nDst);
13 return G_r;
```

(c), statement pairs (a_1, b_1), (a_3, b_3), (a_4, b_4) and (a_5, b_6) are *identical* since they invoke the same Android APIs with the same parameter type *String*, regardless of the *String* value. (a_2, b_2) is not identical as they invoke different Android APIs. Additionally, b_5 is optional and unmatched to any statement in the first method, denoted as (ε, b_5).

Lines 6-12 in Algorithm 1 add the original control flow relationship between statements existing in G to the corresponding statements in graph G_r. Finally, in G_r the vertex is either identical (common amongst cloned methods), or optional (different amongst cloned methods). G_r retains the original control flow relationships. At line 9, $findNode(G_r, src)$ is to find the node in G_r which contains src. As shown in Fig. 3.2(c), the control flow relationship between (a_1, b_1) and (a_2, b_2) is added back. Similarly, all the other relationships between statements in G can be recovered.

3.5.3 Semantic Model Construction

3.5.3.1 DSA Construction

After obtaining differencing graphs for summarized cloned methods, we construct the corresponding DSA. The conversion from a differencing graph to a DSA is a *line-digraph* problem that has been nicely addressed in [94]. Statements with summarized

differences are represented with variables in DSA. In Fig. 3.2(c), the matched statement pair (a_2, b_2), in which a_2 and b_2 are not identical, can be represented as variable x—invoking either getDeviceId or getLine1Number. As there is no matched statement in the first method for b_5, variable y is used to denote either an optional invocation of setRequestProperty or *null* operation (ε).

To capture the essence of malware, we apply two filters to remove the statements regardless of their summarized differences during the construction of DSA. First, we filter out the statements that contain no invocation or branch information, such as int a=10 and return a. However, the statements with branch information like if (a>10) will be kept. The rationale to remove these statements is that—they have no direct interaction with the Android system or the external environment, which means that it can only operate its enclosed data and internal logic in apps, but not directly impact on Android and the external environment [44]. Second, we filter out the statements invoking methods that are not Android APIs. Thus, invocations of methods from third-party libraries are pruned. As the detection is Android-API name sensitive (§ 3.6) and methods from third party libraries are easily obfuscated (e.g., Proguard [95]), we keep invocations to Android APIs that cannot be easily altered.

3.5.3.2 Object-based Action Extraction

To provide a fine-grained model for malicious behaviors, we perform an *OBA extraction* on the DSA. One malicious behavior can be split into several atomic actions, and any two actions may have data flow dependency in between.

As illustrated in Fig. 3.2(d), the behaviors represented by the DSA invoke 5 or 6 operations. Intuitively, these operations can be roughly grouped into two OBAs with meaningful semantics—*obtain sensitive information* and *send out the information*. Specifically, the former action is realized via calling the method getDeviceId or getLine1Number in class TelephonyManager; the latter calls a sequence of methods $\langle$openConnection, setRequestProperty^{+},connect$\rangle$ in class HttpURLConnection. For these API usage patterns related to a specific object, we call them object-based actions (OBAs). In addition, to obtain a complete lifecycle for an object, we consolidate the creation and invocations of

Algorithm 2: Objected-based Action Extraction

Input: D is a DSA
Output: $OBAs$ is a list of OBA, initially empty.

```
1 for node ∈ D do
2     for initialized object obj ∈ node do
3         partial DSA d = forward_slicing(obj);
4         OBA = {d_s | node ∈ d ∧ node.contains(obj)};
5         OBAs = OBAs ∪ OBA;
6 return OBAs;
```

this object. In Fig. 3.2 (a), statement a_1 returns the global instance of TelephonyManager. Combining the creation (a_1) and the invocation (a_2) of the object, we obtain a complete OBA for TelephonyManager.

Algorithm 2 depicts the process to extract actions from DSA. First, we traverse the nodes contained in DSA (line 1). OBAs usually begin with an initialization, i.e., the aforementioned creation of an object, which can return an object of a certain class (line 2). For example, the statement tm=getSystemService() in our running example. Then, we use *forward slicing* [96] to get a partial DSA of which nodes contain the object (lines 3, 4). For example, the invocation of the method getDeviceId in TelephonyManager is categorized into TelehonyManager-based actions. Last, Algorithm 2 returns a list of OBAs, e.g., two OBAs (one TelephonyManager-based and one HTTP-based) are returned for the running example.

3.6 Malware Detection and Classification

The learned DSA can be utilized as signatures (in the form of directed graphs) for malware detection and classification. However, the direct searching based on DSA matching is not scalable [44]. In this section, we propose a combined approach that fast filters out benign apps via machine learning, and then conducts DSA inclusion check on the suspicious apps.

3.6.1 Machine Learning Based Detection

To enable fast malware detection, a machine learning classifier is trained to detect malware. Particularly, two types of features we use for machine learning are listed as follows:

Feature used in previous studies. Basically, features derived from *Risky Android API* (denoted as **feature set *F1***) and *Bigram call sequence of risky APIs* (denoted as ***F2***) are widely used in existing malware detection on Android.

Risky Android APIs refer to APIs with high risks if not properly used by users. Usually, the invocations of these risky Android APIs potentially lead to certain malicious behaviors without the awareness of users. We select 469 risky Android APIs [97] that are frequently used in malware, e.g., `getLine1Number` to access phone number and `sendTextMessage` to send an SMS message. In Table 3.2, 189 out of 469 Android APIs appear in the malware code and constitute *F1*.

Bigram call sequence of risky APIs refer to 2-length call patterns of risky APIs that highly likely appear in malicious behaviours. For example, malware Zitmo and benign app Wechat both register a `BroadcastReceiver` to receive incoming SMS messages. Then Zitmo sends messages to a remote server, while Wechat only shows messages to the local user. Inspecting call sequences of APIs provides more information on the intention of behaviors, and distinguishes malicious apps from benign ones. Hence, we perform an inter-procedural analysis to extract bigram call sequence as *F2*. In Table 3.2, out of all bigram call sequences of the 189 risky APIs, 5923 bigram features appear in malware code and constitute *F2*.

Selecting features according to DSA. To guarantee the comprehensiveness of features, we first add more APIs into *F1* – 182 APIs which do not acquire permissions but are commonly used in malware (e.g., `getContentResolver`), 147 APIs that are related to Java *reflection*, and 2 APIs that are used to invoke native code. Second, we extract more abundant bigram call sequences of APIs. However, using all features of APIs and bigram call sequences leads to the problem of *curse of dimensionality* [98]. To mitigate

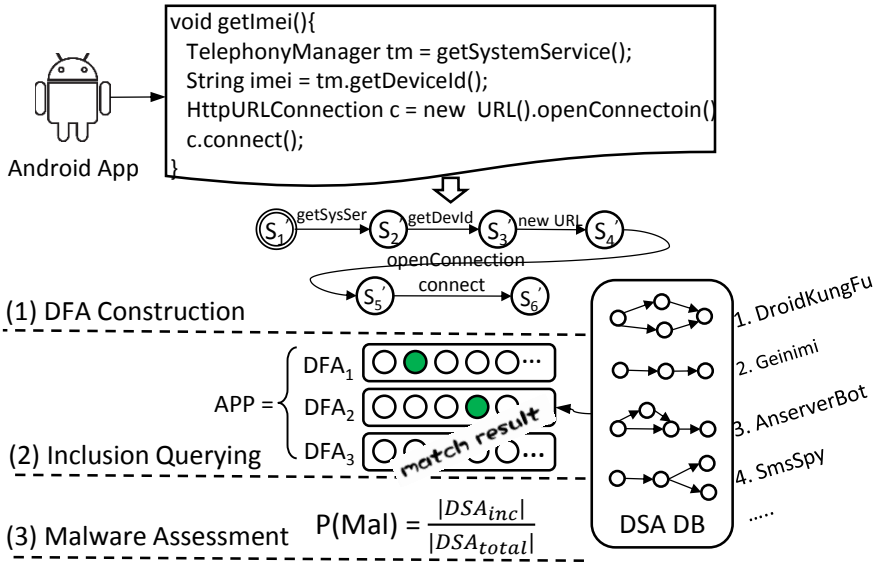

FIGURE 3.5: The process of DSA based detection

this problem, we select APIs and bigram call sequences existing in DSA as ***F3***. In Table 3.2, compared with *F1&F2*, *F3* contains the features and bigram features of the 182 extra APIs that we consider. Note that each feature in *F3* is required to be contained in at least one DSA.

After specifying feature vectors, we apply machine learning to train a classifier which can distinguish malicious apps from benign ones. We employ the *Random Forest* classifier [99] (it achieves the best classification result based on our experiments in Section 3.7.2) to gain a well-trained model from the training set. According to classification results, we can determine if one candidate app is benign or suspicious. For the suspicious apps, we will conduct the DSA based detection to confirm, classify them into known malware families, and identify the attacks involved.

3.6.2 DSA based Detection and Classification

The list of suspicious apps resulting from machine learning step contains false positives, which requires further scrutiny. Thus, to confirm these suspicious apps, we propose the DSA based static analysis, which proceeds in three phases: 1) *DFA construction*. For a given Android app, we analyse and construct one DFA for each method contained in this app. As shown in Fig. 3.5, method `getImei` in this app is converted into a DFA,

illustrating the control flow of the method. In this way, we can obtain a set of DFA for an app; 2) *Inclusion querying*. The extracted DFA will be sent to inquiry the DSA database. In this thesis, we perform a DSA inclusion algorithm to determine if any DSA of malware includes this DFA. If one of the DFA in the app is identified as being included (see Definition 6) in any DSA in the malware family A, it is called a *match* to A; 3) *Malware assessment*. We identify the number of matches as well as its proportion for each malware family, and determine that the app is malicious if the proportion is larger than a certain threshold (denoted as θ_2).

DSA inclusion is to check if all accepted paths in a given DFA are also accepted in the DSA. We define it as follows:

Definition 6. A DFA dfa is included in a DSA dsa **iff** $traces(dfa) \subseteq traces(dsa)$.

Given one DSA, the inclusion check answers whether all paths in the DFA can be found in the DSA or not. We adopt the anti-chain algorithm [100–105] to check whether a DFA is included in the DSA (more details in [106]). For a method from the suspicious app in the previous step, if the DFA in Fig. 3.5 fails to match any learned DSA, it appears that the suspicious app may not reuse the existing attack behaviors or fall into an existing malware family. For further scrutiny, we check if the DFA complies with one or multiple OBAs via OBA inclusion (i.e., checking whether the DFA is included in the OBA, as an OBA is a partial DSA). Consequently, we identify the attacks that highly co-occur with the matched OBAs. In Fig. 3.5, the DFA matches two kinds of OBAs, i.e., TelephonyManager-based action to access IMEI code and HTTP-based action to transfer information, suggesting a potential *privacy leakage* attack.

3.7 Evaluation

SMART is implemented in Java with 10K+ LOC. The experiments are conducted on a Ubuntu 14.04 desktop with Intel Xeon(R) CPU E5-16500 and 16G Memory. Details on SMART and experimental data are available in [106]. The data sets used in our experiments are as follows:

- **Malware benchmark for training (D1)**. We use the latest DREBIN [20] malware collection containing 179 malware families and 5,560 malware apps. Note that DREBIN includes the famous malware collection GENOME that contains 49 malware families and 1,260 samples.

- **Real-world Android apps (D2)**. We crawled totally 223,170 apps from Google Play and 16 popular third party marketplaces. These marketplaces are deployed either in US or China. These apps are collected from 2013 to 2016.

- **Benign apps for training (D3)**. We select 5,600 apps in Google Play, on which 99.8% of available apps are benign according to [107]. We verify the benignity of these apps based on the report of VIRUSTOTAL [60] and collect 5,560 benign ones.

Note that $D1\&D3$ constitute the labelled training dataset used in Table 3.2 and Table 3.3; $D2$ is the unlabelled dataset used for wild prediction on unlabelled apps in Table 3.4. The experiments are conducted to evaluate our approach in terms of five aspects explained in the subsections.

3.7.1 RQ1: Evaluation of the Semantic Model

Controlled experiments on the data set of DREBIN [20] are conducted to evaluate the usefulness of our semantic models. For each malware family, we generate a set of DSA and extract the corresponding OBAs to characterize attack behaviors. On average, there are 101 DSA learned for each family, which denotes the average complexity of attacks involved. There are around 20 nodes (i.e., code statements) on average in one DSA, containing about 6 variables that occupy 30% of all nodes in one DSA. In addition, we identify 20 types of OBAs with 193 variants existing in DREBIN. See [106] for detailed models.

Soundness & completeness. To justify the soundness of the DSA models, we select a number of samples for each malware family in DREBIN. We omit the families with only one sample, as a DSA generated from one sample is identical as the corresponding DFA.

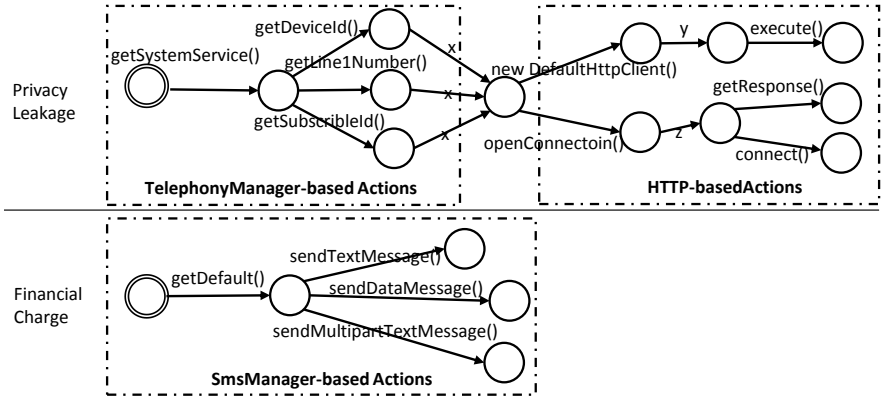

FIGURE 3.6: The DSA and actions for common attacks

We find that all of the selected samples are correctly detected as malware. It proves that the constructed DSA are able to capture the essential parts of known malware. With an in-depth inspection of constructed DSA of 179 malware families, we confirm that the set of DSA for each malware family contains at least one malicious behavior. This proves the completeness of learned DSA— each malware family has malicious behaviors modelled as DSA. Moreover, due to the filtering mechanisms in bytecode clone detection in Section 3.5.1, we may omit a few malicious behaviors existing in the minority of variants. However, the common malicious behaviors are retained in the form of DSA.

As the OBAs are extracted from DSA and related to the "assets" users possess, they are capable of composing attacks. The two examples of attacks in Fig. 3.6 demonstrate the soundness of OBAs. The attack *privacy leakage* contains two basic operations: access sensitive information and send out the information. According to the result of OBA extraction, we find that the attack of stealing phone's profile can be implemented via TelephonyManager-based and HTTP-based actions. Similarly, the attack *financial charge* only involves one type of actions, i.e., SmsManager-based actions. As 20 types of OBAs are verified to be related to the assets exploitable by attacks, the OBAs cover the 6 common attacks (See [106]) in DREBIN, indicating the completeness of OBAs.

Observations of malware evolution. Since DSA is a comprehensive summarization of malware variants, we observe malware evolution from the generated DSA. Specifically, two ways of malware evolution or mutations to variant are observed: 1) In many malware families, variants are evolving in non-functional content, such as parameters (e.g.,

the address of remote server and the number of receiving phones) — malware variants of *parametrised clones*; 2) Variants in some malware families are enhanced by supporting more similar operations — malware variants of *gapped clones*. For example, Zsone registers a BroadcastReceiver to monitor the reception of SMS, and cancels the broadcast if the message comes from premium numbers. Meanwhile, it keeps adding new premium numbers in its variants. Most of this type of variants can be captured by our clone detection.

Summary. DSA can effectively model and help to detect the malware variants with exact clones, parameterised clones and clones containing small gaps. However, in our evaluation, we find one FN case missed by SMART—replacing functional code by equivalent code (e.g., DroidKungFu uses Socket to replace DefaultHttpClient to perform the communication operation) or using advanced obfuscation (e.g., String encoding, reflection and native code).

3.7.2 RQ2: Malware Detection based on ML

In this experiment, we select 5,560 malware samples in DREBIN and 5,560 benign apps from Google Play as our data set, and evaluate the results of different classifiers and the enhancement via DSA of machine learning based malware detection. All experimental results are obtained by performing 10-fold cross validation (CV).

The accuracy of the detection classifiers. To evaluate the accuracy of our training model, we extract features *F3* (§ 3.6.1) from DSA in our data set. We compare these classifiers: *AdaBoost* (87.8%), *C4.5* (93.8%), *Linear SVM* (90.4%), *Naïve Bayes* (81.6%) and *Random Forest* (97.0%). It is concluded that Random Forest achieves the best classification result. Hence, we employ Random Forest to train our model.

DSA-enhanced classifier. We set up a comprehensive experiment to investigate the usefulness of different feature sets and DSA in the classification. Table 3.2 shows that feature set *F3* achieves the best result with the Random Forest classifier. As introduced in Section 3.6.1, feature set *F1* derived from risky APIs is used in recent study [53].

TABLE 3.2: Accuracies of classifiers using different feature sets on the training dataset that contains DREBIN

Feature Set	# Features	Precision	Recall	F-Measure
F1	189	92.4	92.3	92.3
F2	5923	95.2	95.0	95.0
F1 & F2	6112	95.4	95.3	95.3
F3	12514	97.0	97.0	97.0

Feature set *F2*, which also has features derived from bigram call sequences, is evaluated in the state-of-the-art tool [108]. Thus, feature set *F3* derived from our DSA is more effective in detecting malware than feature sets *F1*, *F2* and its combination. The rationale is that *F3* excludes many bigram call sequences that do not relate to malicious behaviors, and moreover, provides significant features from DSA such as reflection and invocation of native code.

Moreover, we list significant features which can effectively distinguish malware from benign apps in Table 3.3. The first two columns in Table 3.3 show the most remarkable features as well as their weight in the classification extracted from DSA. 3 out of top 5 features are related to the access to local storage, which are commonly used in Android malware to dynamically load payload (e.g., BaseBridge), store sensitive information (e.g., DroidKungFu), which are rarely used in benign apps; and the second two columns show features extracted from the whole code. In these five remarkable features, bigram call sequences pause $\rightarrow$ start and setVideoPath $\rightarrow$ pause are related to the operation of video player in Android. Generally, they are not considered the components of malicious behaviors. As a result, the top features extracted from DSA are more reasonable and representative for malicious behaviors (see the full list of weights of features in [106]); the third two columns show the significantly different features and their differences between these two experiments.

Summary. In the 10-fold CV training, SMART achieves the highest accuracy (97%) when using *F3* with *Random Forest*, by virtue of the feature selection based on the DSA.

TABLE 3.3: Top features w/o DSA of machine learning

Features in DSA		**Features in code**		**Most different features**	
Features	Weight	Feature	Weight	Feature	Difference
getCacheDir	0.19	getDatabasePath	0.22	getCacheDir → getDatabasePath	-0.20
getContentResolver → openInputStream	0.13	getCacheDir → getDatabasePath	0.21	getDatabasePath	-0.18
getPackageManager → getContentResolver	0.13	getContentResolver → openInputStream	0.17	setVideoPath → pause	-0.16
openInputStream	0.10	pause → start	0.17	pause → start	0.16
openXmlResourceParser	0.07	setVideoPath → pause	0.16	getPackageManager → getInstallerPackageName	0.09

3.7.3 RQ3: Evaluation on Real World Apps

We collect 223,170 free Android apps from Google play and third-party Android markets. The data sets are listed in Table 3.4 with the corresponding total number, and the number of suspicious apps detected by SMART as well as prestigious anti-virus (AV) tools, including AVG (AG), Avast (AT), BitDefender (BD), F-Secure (FS) and Sophos (SO). As shown in Table 3.4, SMART can detect 4,583 (2.1%) out of all collected apps based on machine learning (ML), while anti-virus tools only regard totally 527 (2.5%) of the 4,583 as malware. With the DSA inclusion check (DSA), SMART rightly classifies 527 (out of 4,583) samples into 23 known malware families — this implies that the approach based on DSA has consistent results with that based on ML, while ML reports more suspicious ones. According to DSA inclusion check, about 2.5% of the malware are classified into existing malware families from DREBIN, while others are not due to the timeliness of DREBIN. Our training data set DREBIN is mainly collected from 2010 to 2013, while real-world apps are collected from Dec. 2013 to Jan. 2016. Plenty of known malware had been removed from each marketplace. Still, among the detected variants detected by DSA inclusion, we identify 22 variants of family Boxer and 12 variants of family Foncy, both of which send unauthorized SMS.

We perform attack identification for the suspicious apps (reported by ML) via extracting OBAs. We identify the 5 most popular actions employed by these apps, which are TelephonyManager-based actions (89.9%), HTTP-based actions (71.6%), ContentResolver-based actions (66.1%), AssetManager-based actions (53.9%), and SmsManager-based actions (40.0%). We have manually checked 100 apps, and 76% of them are found containing malicious behaviors, which are mainly privacy leakage and sending SMS message to premium numbers. Taking the app “Super SMS Quick Delete” from

TABLE 3.4: The data sets for evaluation on real-world apps

Marketplace	**# App**	**SMART**		**Anti-Virus Tools**				
		ML	**DSA**	AG	AT	BD	FS	SO
AndroidDrawer [109]	11731	200	5	246	152	200	212	187
AnZhi [110]	46757	848	34	702	613	276	300	796
Apkmirror [111]	23441	165	11	201	180	79	165	88
AppsApk [112]	2481	86	26	40	23	8	12	43
ChinaMobile [113]	1714	61	10	384	118	98	102	443
Coolapk [114]	19969	452	25	402	241	40	80	242
Eoemarket [115]	5895	101	3	953	346	433	606	780
FDroid [116]	4533	139	29	10	8	3	5	3
Flyme [117]	10927	614	121	1281	122	305	488	366
GetJar [118]	42633	498	89	2322	211	205	1233	1055
GFan [119]	1000	75	7	89	65	70	62	54
Google Play [120]	7643	45	37	220	81	55	89	41
Huawei [121]	9856	275	49	1730	1128	526	679	828
SlideMe [122]	5770	63	8	866	523	325	124	324
Wandoujia [123]	3066	163	20	189	50	120	156	62
Wangyi [124]	7272	120	8	138	142	98	86	87
Xiaomi [125]	18482	678	45	2497	999	1005	1332	1539
Total	223170	4583	527	12270	5002	3846	5731	6938

AppsApk for example, we extracted OBAs existing in this app, and found it had TelephonyManager-based actions, with which it can access IMEI, IMSI, etc. In addition, it had SmsManager-based actions which can send SMS messages via sendTextMessage, as well as HTTP-based actions. It is concluded that this app can expose sensitive information to attackers. All the analysis results can be found at [106].

Summary. The ML (using *F1&F3*) and DSA inclusion in SMART detect more malware variants than the AV tools. Compared with ML, DSA inclusion is susceptible to the timeliness of the training set for family identification. However, it is more tolerant of malware variants. In contrast, AV tools detect more malware while many of them (76.6%) belong to *Adware*, which are not considered by SMART.

TABLE 3.5: The capability of detecting malware variants

Malware Variants	SMART	Anti-Virus Tools				
		AG	AT	BD	FS	SO
1. Repacking	✓	✓	✗	✓	✓	✗
2. Dis- and Re-assembling	✓	✓	✗	✓	✓	✗
3. Changing Package Name	✓	✓	✗	✓	✓	✗
4. Identifier Renaming	✓	✓	✗	✓	✓	✗
5. Data Encryption	✓	✗	✗	✓	✓	✗
6. Call Indirections	✓	✗	✗	✓	✓	✗
7. Code Reordering	✓	✗	✗	✓	✓	✗
8. Junk Code Insertion	✓	✗	✗	✓	✗	✗
9. Function Out- and Inlining	✓	✗	✗	✓	✓	✗
10.Other Simple Transformations	✓	✗	✗	✗	✗	✗
11.Composite Transformations	✓	✗	✗	✗	✗	✗
12.Remove Malicious Code	✗	✓	✓	✓	✓	✓
13.Construction based on DSA	✓	✗	✗	✗	✗	✗
Detection Rate (%)	100	30.8	0	69.2	61.5	0

3.7.4 RQ4: Resilience to Malware Variants

To validate the resilience of SMART to the transformation attacks [126] and advanced variants, we compare the detection rate of SMART with other AV tools on manually crafted variants of known malware. The malware variants are crafted in three ways: 1) employ transformation attacks [126] to generate malware variants for testing (variants 1-11); 2) remove the malicious code from malware to test the false positive rate (variant 12); 3) craft malicious apps in terms of DSA and actions summarized from the malware collection (variant 13). The variants are published at [106].

As shown in Table 3.5, SMART exhibits the highest detection rate, as our combined detection approach (ML+DSA) considers the essential part of malware that are unchanged in these variants. AV tools achieve overall 32.3% detection rate of these variants. Note

that the tool SO, which detects 79 real-world malware, fails to detect any crafted variants. By careful inspection of its mobile app, we infer that it relies on exact matching of signature of malware. Besides, we learn three kinds of features utilized by AV tools as follows:

- **Non-code files**. Specifically, the file *AndroidManifest.xml*, which declares acquired permissions, activities, services, etc., is scanned. BITDEFENDER can resist two complex transformations (variants 8, 9) when retaining AndroidManifest.xml, but fail when removing it.
- **Structure of code files**. AV tools check the structure of classes and methods in an app. Thus, if we keep the structure of the original app, however, remove the malicious code inside, these AV tools produce false positives.
- **Lexical features of code**. We create a new variant (No. 13) by following the DSA in malware family SMSSpy, which differs from the original malware in lexical features. None of AV tools is capable of detecting it.

Summary. SMART is capable in identifying malware variants according to transformation attacks. Variants generated according to DSA, e.g., 4 variants derived from Fig. 3.2(d), can bypass the above three strategies of AV tools.

3.7.5 RQ5: Scalability & Efficiency

We measure the performance for the two phases of SMART, the offline model learning, and online malware detection and classification.

Offline model learning mainly consists of two computations: 1) typed 3D-CFG generation and 2) clone differencing and DSA generation. Given categorized malware data, we only need to perform model learning once. For each family of malware in DREBIN, the average time for clone detection is 72.5s, and in total it takes around 3.6h for 179 families. The clone differencing and DSA generation needs 167.5s on average. Although it takes a long time (around 19 days) to generate all 18,000 DSA, it only needs to be done once and the computation can run in parallel using multiple machines.

Online malware detection consists of two steps: 1) ML based malware detection and 2) DSA based inclusion for family classification. To check if an app is malicious, the ML step takes 13.4s on average (13.3s for feature extraction and 0.1s for prediction). If the app is detected as malware, it takes 105.9s on average to perform DSA based classification to confirm and classify it according to the 18,000 DSA in our database. Assuming the probability of an app detected by ML is 0.03 (§ 3.7.3), the expected time for the detection and classification is 16.6s for each app, which makes SMART efficient for a large-scale scan.

3.8 Discussion

Threat to validity. Threats to *internal validity* come from two thresholds used in bytecode clone detection θ_1 in Section 3.5.1 and malware assessment θ_2 in Section 3.6.2. A high value of θ_1 leads to few accidental clones but also few valid clones that are partially shared by malware variants, and vice versa. θ_2 intuitively specifies the maximal tolerance to the mutation of malware. If θ_2 is high, the DSA based approach can achieve more accurate results while tolerate less mutations of variants, and vice versa. Threats to *external validity* are two-folds: the *timeliness* of malware dataset and the *deficiency* of static analysis in our machine learning approach. Since the new malware is emerging every day and our DSA based approach performs an accurate matching with known malware, it can degrade the result if apps contain new malware that is not included in the known malware collection; Similar to [21], we do not carry out a complete static analysis to elevate the performance in ML approach. Inherently, static analysis is susceptible to dynamic loading and reflection, which is another threat to external validity.

Sufficiency of DSA in modelling Android malware. According to [5], malware samples in the same family carry similar malicious behaviors, which lead to clones involving similar attack targets and manners. Hence, malicious behaviors can be well captured with clone detection, and DSA can present the essential parts of malicious behavior while remain the variety of malware variants. Second, Android attacks usually call the OS built-in API. Thus we model the API call as the transition in DSA. The

new attack that dynamically loads malicious payloads may not have the oblivious behaviors or DSA [57], but it still relies on some reflection APIs to conduct loading. Last, to avoid the path explosion problem, condition guards (arguments of API invocation) are not considered in the behavior model of DSA or DFA. Similarly, existing work on Android API dependency graph does not consider invocation arguments.

Usefulness of DSA in malware detection. As DSA can capture the common malicious behaviors, we use the learned DSA to guide the feature extraction in DREBIN (i.e., only extracting the concerned features included in DSA). Even using thousands of extra features derived from the 182 extra APIs (§ 3.5.3), the ML training stage becomes more efficient with 10% reduction of running time yet without affecting the accuracy (both as high as 98%). Besides, compared with other tools, SMART has the advantage in finding fine-grained attack actions, representing them as OBAs based on analysis of source-sink path [21] and control flow graph [43].

Enhancement of ML based malware detection. Features of bigram call sequences of Android APIs, especially those contained in DSA, can improve the detection accuracy of the ML based approaches by 5%-10%. As bigram call sequences retain the direct relationship between APIs, it can effectively depict coarse grained malicious behaviors [21, 44]. We limit the n-gram analysis of call sequence to bigram due to the efficiency issue. Note that the results of evaluation of SMART on testing datasets show that the introduction of bigram features does not cause overfitting problem, as the FP (13.7%) rate of SMART is still low.

3.9 Conclusion

In the chapter, we built a hierarchical semantic model for Android malware. We developed a framework, named SMART, to automatically learn models from malware, and use a combined approach of machine learning and DSA inclusion to detect and classify malware. The results of experiments show that our approach can achieve both efficiency and scalability.

4

Evolving Android Malware for Auditing Anti-Malware Tools

4.1 Introduction

Malware detection is always one of the central topics in cyber-security. Anti-malware tools (AMTs) are getting more advanced, but as a result surviving malware is getting increasingly sophisticated. Generally speaking, the development of AMTs usually lags behind the advance of new malware, since new *malware variants* (similar malware generated by software obfuscation or configuration techniques) and zero-day *vulnerabilities* (a weakness that allows an attacker to exploit the target system) keep emerging every day.

In retrospect, Android malware undergoes a stunningly rapid increase in a short time of last five years. In 2010, we witnessed the industrial age of mobile malware, and also in that year Geinimi was one of the first found malware that attacked the Android platform and used the infected phone as part of a mobile botnet [127]. Ever since then, a larger number of sophisticated mobile malware is created by attackers due to the prevalence of Android phones. With regard to the defence side, traditional approaches relying on textual or hexadecimal signatures [128] are incapable of detecting variants of existing malware and new ones. To catch up the trend, in research community, machine learning based approaches [20, 21, 44, 53] and information-flow analysis based approaches [22, 23, 40] are proposed to detect obfuscated malware and their variants. Recently, new attacks (transformation attacks [126, 129] and collusion attacks [128, 130]) are revealed, and they fail the existing detection approaches. Thus the similar arms race between malware and anti-malware is unexceptionally observed on Android platform.

Recently, Android malware exhibits a variety of attack behaviors, including leaking user privacy, escalating privilege without permission, conducting unknown financial charge, and abusing application functionality. In real malware, these attack behaviors may co-exist in order to increase the damage and success probability of the attack. Even worse, evasion techniques (e.g., multiple-level obfuscation [126, 129]) are further applied on the code or deployment package to evade the scanning of AMTs. Hence, the combination of different attack behaviors and evasion techniques indeed exist in real-world Android malware [5] — such a fact hinders the understanding of the reason why AMTs fail in detection.

The main challenge in auditing the AMTs is the lack of the evaluation criteria and well labeled benchmark such that we cannot evaluate and the strength and weakness of AMTs systematically. To our best knowledge, existing AMT evaluation is based on benchmarks like GENOME [5] and DREBIN [20], which classify malware based on families only. GENOME and DREBIN are not suitable for auditing AMTs due to three reasons: 1) the malware are old and well recorded in malware repository of AV tools, 2) there is no comprehensive coverage of different attacking and evasion techniques, 3) there is no index (or label) to different malware attributes (like attack behavior, obfuscation techniques, and anti-debugging techniques), except for (inaccurate) family names.

By far, only one existing work [126] discusses the resilience of different *Defence strategys* (DS) against the obfuscation techniques.

In this thesis, we propose to separate different attack behaviors and evasion techniques into basic reusable features — to summarize the attack features and evasion features of malware on Android. Here, *attack feature* (AF) means malicious behavior of a certain attack, which links to implementation of the functional requirements (intention) of malware. *Evasion feature* (EF) means the ability of malware to evade the scanning of AMTs, including a variety of code complication and transformation techniques that change no functional requirements of malware. In this way, we can develop a meta model for Android malware by modularizing various attack features into the various *building blocks*. Hence this meta model allows us to generate malware variants to cover different AFs and EFs, and evaluate how different DSs react to each individual feature as well as their different combinations.

Technically, we start with detecting and analyzing the malicious code among the similar yet different malware variants in an Android malware sample. We then modularize the malicious code from different malware families into different AFs, and identify evasion techniques into EFs. Once features are modularized, we adopt the concept of Software Product Line Engineering (SPLE) and build a feature-oriented architecture [131] as the malware meta model to capture the different features and their constraints inside malware.

With the malware meta model, we apply a multiple-objective evolutionary algorithm (MOEA) to mimic the evolution of the malware. MOEA performs *gene crossover* (i.e., exchanging features of two samples) and *mutations* (i.e., selecting or deselecting feature under mutation) on the current malware generation to produce next malware generation. To guide the evolution to generate *better* malware, we define the fitness function for selecting next generation by maximizing the number of attack behaviors, minimizing of evasion techniques needed and the expected detection rate. To automatically generate malware, we propose an intermediate behavior modeling language, *Behavior Description Language (BDL)*, to perform model-driven malware generation by gluing the code snippets of different features. In this way, we can generate the malware from simple

(single AF) to complex (combinations of AFs), from plain malware (without EFs) to highly evasive ones.

Finally, we develop the proposed malware generation process into a tool called MYSTIQUE, and use it to audit 57 off-the-shelf anti-malware tools and 9 academic solutions in terms of detection ratio and capability with 10,000 generated malware. With the experiment results, we test four commonsense hypothesis of AMTs. To check the capability of online vetting of app stores, we upload 12 generated malicious apps onto 4 mainstream app stores. In most cases, our malware passes the online vetting process. Furthermore, we propose some possible enhancements for the existing malware detection approaches. To sum up, we make the following contributions:

- We recognize the Android malware as attack features and evasion features and present them in a meta model. Different from focusing on requirements in malware ontology analysis [132], we consider and maintain the traceability between the features and their corresponding code in MYSTIQUE.
- Based on the meta model, we develop an SPL architecture to generate new Android malware by an MOEA. Our approach is implemented in an automated framework named MYSTIQUE. In malware generation, we propose a BDL to support model-driven code assembly and verify constraints in code assembly.
- We survey and evaluate the state-of-the-art AMTs using our generated malware. The experiments show that the existing AMTs are quite weak at detecting these new malware — a detection ratio of less than 30% on average. We propose the countermeasures in order to detect the malware generated by MYSTIQUE.
- We have generated over 10,000 samples of Android malware by combining different attack and evasion features. They can serve as a benchmark to assess detection capabilities of AMTs, as they cover different representative combinations of features, which are missing in the current malware benchmarks.

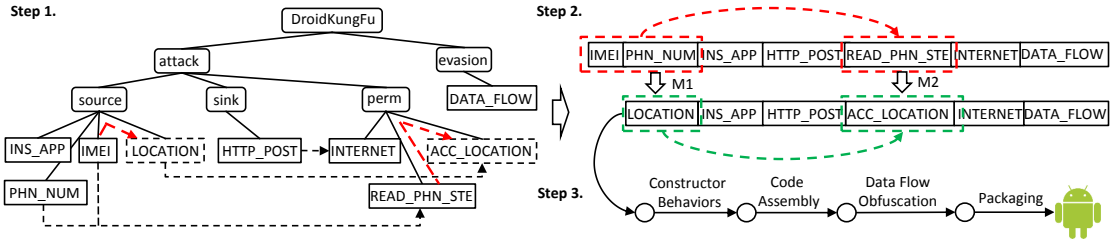

FIGURE 4.1: A running example to illustrate the generation of new variant of DroidKungFu

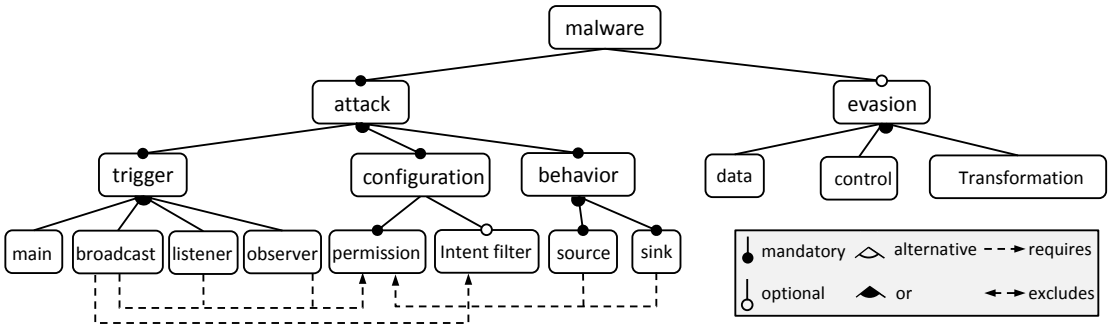

FIGURE 4.2: The partial feature model of privacy leakage malware

4.2 Mystique Overview

This section explains the high-level idea of our approach MYSTIQUE and depicts the major steps of our approach with a running example. We also list the potential challenges in this study.

4.2.1 Mystique Overview

We show the overview of the malware generation using Fig. 4.3. The input of MYSTIQUE includes the original malware collection (i.e., GENOME [5] in this work), and the identified meta model with the code snippets of features. The output is a collection of generated malware samples that are labeled with features inside.

Here, we briefly describe the work flow. First, the attack features (AF) and evasion features (EF) used in the original malware collection should be identified and modularised (§ 4.3.1). Currently, the identification is aided by bytecode clone detection and the modularization is manually done [133]. Attack features are the requirements of malware from the prospective of malware creators, and atomic functionalities from the

prospective of SPL. They are the fundamental composition of activities (i.e., behaviors) of Android apps. After finishing the feature-oriented domain analysis (FODA) for the 49 malware families in GENOME, we remove the duplicated AFs and add the remaining AFs in the feature model shown in Fig. 4.2. Meanwhile, the corresponding code of these AFs are also modularised in separated methods or classes, in such a way that the architecture of our malware product line is built up. EFs are mainly from evasion techniques in flow and code. EFs in flow include multiple ways from source to sink, such as Lifecycle [22] and ICC [39], and EFs in code are inspired by the transformation of code [126, 129]. Once the features in our FODA are available and the corresponding feature code are modularized, we adopt the Indicator-Based Evolutionary Algorithm [134, 135] to generate malware with the three objectives: *aggressiveness*, *evasiveness* and *detectability* (§ 4.4).

Running Example To illustrate the process of the construction, we take *DroidKungFu* in GENOME as an example. *DroidKungFu* belongs to malware of privacy leakage, which steals sensitive information (e.g., IMEI code, phone number). Fig. 4.1 shows the feature model for malicious behaviors contained in *DroidKungFu*. The feature model consists of necessary features for launching attacks as well as their relationship in between. For example, *DroidKungFu* obtains IMEI code and phone number, which belong to the feature Source. According to Fig. 4.2, the mandatory feature Source and Sink must be selected as they are both needed for privacy leakage. Feature Source has 4 optional sub-features, among which feature IMEI and PHN_NUM require permission feature READ_PHN_STE that relates to permission (§ 4.3.1).

After obtaining the feature model, we encode the attack features (malicious behaviors) as chromosome in gene. Specifically, each bit of the chromosome represents the existence of an attack feature — 1 for existent and 0 for non-existent. The *crossover* operation and *mutation* operations are performed on the chromosome during the evolution. In step 2 of Fig. 4.1, we mutate features IMEI and PHN_NUM to feature LOCATION. According to feature dependency, the original permission feature READ_PHN_STE for IMEI is not required any more, while feature LOCATION acquires permission feature ACC_LOCATION. After that, the gene of a new malware variant is produced.

In step 3 of Fig. 4.1, the code of each feature in gene will be assembled. Details on assembling triggers and manifest file are in Section 4.4. In addition, to audit the capabilities of AMTs, we also employ obfuscation to evade the detection, then one malware variant is created and can be used for auditing AMTs. Owing to the gene of the generated malware variant, we can evaluate what feature combinations can evade what defence strategies.

4.2.2 Technical Challenge

To generate the sound and workable malware, we face the following technical challenges:

- **C1: Construction of feature model.** A variety of features are contained in malware, and used in different ways. It is difficult to identify and extract the representative features in malware. Section 4.3.1 addresses this challenge.
- **C2: Malware measurement.** With the feature model (i.e., the meta-model of malware), we still need some goals to guide automated malware generation. Thus, we define three objectives: aggressiveness, evasiveness and detectability for measuring malware (§ 4.4). Since these objectives are competing (e.g., highly offensive malware is generally more detectable), we use an MOEA to select feature from feature model in Section 4.4.
- **C3: Automated code generation for malware.** A valid malicious app conducts certain malicious behaviors, with no errors during compilation and runtime. Between feature model and the corresponding workable implementation, there is still a big gap. Hence, in Section 4.4, we mitigate the challenge of gluing code of features into valid malware via model-driven code generation.
- **C4: Validation of generated malware.** As the malware is generated according to the feature model, we want to prove their maliciousness — whether malicious behaviors can be triggered and carried out on real Android devices. To address this challenge, in Section 4.5, we conduct the following experiments. For the attack of privacy leaks, we set up a dummy server or device to receive the sensitive information sent from malware.

4.3 Feature-oriented Domain Analysis of Android Malware

We identify the common building blocks for Android malware as *attack features* and *evasion features*, and propose to use feature model as the meta model to capture the malware. Different from focusing on requirements in malware ontology analysis [132], we consider and maintain the traceability between the features and their corresponding code in our model. One partial feature model of malware of privacy leakage in GENOME is shown in Fig. 4.2[1]. Note that feature-oriented domain analysis relies much on the domain knowledge of security experts, and only feature and their code relevant to attacks are manually modularized. We totally identify 266 attack features and 14 evasion features. Note that our feature model is not a complete one, but can be extended to covering new attack behaviors and evasion methods.

4.3.1 Attack Feature

Attack features refer to features that are generally relevant to the malicious behaviors of a certain type of attack. They can be further categorized into the following three types:

Trigger Feature. *Trigger* defines the entry points for malicious attack behaviors. Triggers are roughly categorized into GUI-based and non GUI-based [24, 44]. Since the GUI-based triggers are attached to visible GUI components in Android, they are easy to be discovered by end users [136]. In this thesis, we only consider non GUI-based triggers without the interaction with end users. From our observation, four kinds of non GUI-based trigger features are mainly identified in GENOME: main, broadcast, listener and observer. The trigger main denotes that malicious behaviors are triggered from the startup of malware; malicious behaviors can be triggered from a broadcast message, while malware needs to register a BroadcastReceiver; malware can also register a listener to listen the changes on states (e.g., location or phone state); malware can register an

[1]The complete feature model of Android malware is available in our website [133].

observer on a ContentProvider. Once the content provider is changed, malware receives an event and triggers malicious behaviors.

Configuration Feature. Two kinds of configuration features are relevant to malicious attack behaviors in malware: *permission* and *intent filter*. Android provides a permission-based mechanism to avoid the abuse of system sensitive operations, e.g., invoking sensitive Android APIs, accessing the sensitive resource on device. Many malicious behaviors in malware require certain permissions to attain attack goals. For example, it needs the permission android.permission.READ_PHONE_STATE to obtain the IMEI code of the device via invoking the method getDeviceId. Feature Intent_Filter defines the acceptable Intent that can be captured by Android components. For example, if one BroadcastReceiver is assigned with intent filter android.provider.Telephony.SMS_RECEIVED, it can capture broadcast messages indicating an incoming SMS.

Behavior Feature. Attack trigger and configuration features are all assistant to the core attack features — behavior features. In an attack of privacy leakage, there are mainly two types of features: Source and Sink. Source is responsible for stealing sensitive information of device, and sink is responsible for sending out sensitive information. Based on the manual domain analysis of GENOME, 11 kinds of source features and 2 kinds of sink features are identified in Table 4.1.

Note that the partial feature model in Fig. 4.2 mainly illustrates the high-level organization of these features. Each leaf feature in Fig. 4.2 may have several subfeatures, e.g., feature Source has 11 variant sub-features in an *Or* relationship, and each variant feature in Table 4.1 may also have several implementation features (modularized code) in an *Alternative* relationship. Interested readers can refer to Section 4.4 and our tool website [133] for more details.

4.3.2 Evasion Feature

Information flows from *source* to *sink* in privacy leakage attack can be obfuscated in three different ways as follows:

TABLE 4.1: Attack behavior features in GENOME

Source	**Category**
TELEPHONY	IMEI, IMSI, PHONE_NUMBER, etc
SMS	INBOX, INCOMING_SMS, etc
CALL	CALL_LOG, INCOMING_CALL, etc
BROWSER	BROWSER_HISTORY, etc
MEDIA	RECORD_AUDIO, etc
LOCATION	REAL_TIME_LOCATION, etc
BUILD	CODE_NAME, SDK, etc
CONTACT	CONTACT, etc
ACCOUNT	ACCOUNT, etc
STORAGE	EXTERNAL, etc
PACKAGE	INSTALLED_APK, etc
Sink	**Category**
HTTP	APACHE_GET, SOCKET_GET, etc
SMS	SEND_TEXT_MESSAGE, etc

Control based Evasion. Malicious behaviors can be obfuscated by complicating the control flow of the attack. Android provides an amount of callback functions to guarantee implicit control flow. Besides, each component in Android has its own lifecycle together with a set of built-in APIs for lifecycle management. In an attack of privacy leakage, the control flow usually involves the interactions between different components in Android. Thus, it is feasible for the attacker to hide the malicious code into the different stages of components and trigger it under certain scenarios. For example, each Android component has a life cycle, the method invocation sequence of which is defined in the framework layer of Android. A certain method will be invoked if the component is in a specific state. In Inter-Component Communication (ICC), there is also one mechanism for implicit control flow if an `Intent` object is not assigned with a determined class [39, 40]. For example, when an activity is launched, the inherited methods `onCreate`, `onStart` and `onResume` are called in sequence [137]. Thus, code in the three methods has a implicit relationship of control flow.

Data based Evasion. Attacks like privacy leakage must conduct data transmission. Such transmission can happen between different methods, classes, apps, or even different channels (i.e., external persistent storage and memory).

- **Persistent Storage**. On Android, applications may exchange data through persistent storage. There are three types of persistent storage provided by Android: *file, shared preferences and SQLite database*. They can be used for applications or components to exchange data — they provide an implicit data flow from one component to another.
- **Memory**. Data that is temporarily stored in a specific memory location (e.g., an object in the Java heap) might be accessed globally. As a consequence, once there is a component or method to fetch data from that memory location, it establishes a data flow from where the data is stored to where it is fetched.

Both. Attacks can be further complicated by combining the obfuscations on both control and data flow. As Intents are the main vehicle for app communication, they can be used for a purpose of advanced obfuscation. One intent can be either explicit or implicit. Explicit intents have a specific class to start, while implicit intents do not specify the corresponding class, and the system will select the most well-suited class or application to execute. An explicit intent can only invoke a specific component, which is defined in the constructor, or by calling setComponent(ComponentName) or setClass(Context, Class); an implicit intent can be received by many well-suited components. It appoints potential receivers by setting an action in the constructor or setAction(String) (Meanwhile, it can be instrumented with a data type to restrict its receivers). In addition, an Intent object can be bundled with some data by invoking putExtra, which generates a data flow from the caller component to the callee component. Therefore, Intent can influence the execution order (i.e., control flow) of the app and also the data flow if enclosed with *extras*.

In addition, we use the transformation attacks [126, 129] to complicate and transform the source code at the implementation level.

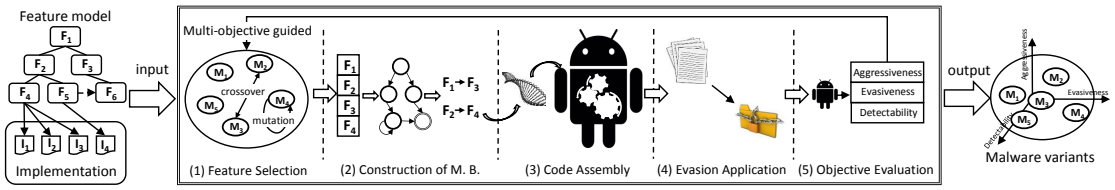

FIGURE 4.3: Multi-objective guided malware generation

Transformation Attacks. We have selected 12 types of transformation attacks, such as identifier renaming, data encryption, code reordering, to obfuscate the generated malware. Different from the previous evasion techniques, the transformations cannot change the behaviors or flows from source to sink. It is a kind of non-behavior evasion technique since the transformations only change the lexical information or code structure. However, by adding a lot of noise to the previous code, it may bypass the detection of AMTs which consider code structure or lexical information. With the reordering of code, it may break some AMTs based on static analysis.

4.4 Multi-objective Guided Malware Generation

This section is devoted to the Android malware generation process in MYSTIQUE. As shown in Fig. 4.3, MYSTIQUE takes a malware feature model as the input and generates various malware variants. During the process, each malware is encoded as a DNA sequence based on their values for AF and EF in the feature model. The malware evolution is an iterative process in accordance with the principle of survival of the fittest, where we randomly choose an initial population of malware satisfying the input feature model, and select better malware in the each generation afterwards based on the fitness function, i.e., the malware measurement function in our case. The five steps in each iteration of malware evolution are listed as follows.

- **Step 1: Feature Selection.** Given the previous generation n of malware, step 1 is to produce new malware by applying crossover and mutation operations on malware from generation n. Considering the three objectives, the IBEA is used to choose the fittest ones in each iteration.

- **Step 2: Construction of Malicious Behaviors.** This step constructs the logic of malicious behaviors, according to the selected features in step 1. To bridge the gap between feature model and code, we present an intermediate language, BDL, to model the logic of malicious behaviors.

- **Step 3: Code Assembly.** To make the malware run on a real device, MYSTIQUE conducts to validate the assembled code and package it into a deployable app. It includes the setup of *triggers* that act as entry points of malicious behaviors, the configuration of manifest file, and malware packaging.

- **Step 4: Evasion Application.** After evasion features are selected, the corresponding evasion techniques are applied. Note that the evasion is based on the source code, without changing the malicious intent or behaviors of the constructed malware.

- **Step 5: Objective Evaluation.** In this step, we calculate the objective functions and choose the fittest for the next iteration. We also need to check whether the evolution is finished due to convergence of EA or reaching the upper limit of iteration times.

The selected features of a feature model is encoded using an array-based chromosome as shown in the step 2 of Fig. 4.1. Given a chromosome of length n, array indices are numbered from 0 to $n-1$. Each feature (no matter AF or EF) is assigned with an array index starting from 0. Each value on the chromosome is z_i such that $z_i \in \mathbb{Z} \wedge z_i \in \{0,1\}$, where 0 (resp. 1) represents the absence (resp. presence) of the feature. Given a feature model M, we define a function $f_M : \mathit{Fea}(M) \rightarrow \{\mathbb{Z}, \perp\}$ that maps each feature f of the feature model M to an array index. $f_M(f_1) = \perp$ denotes that there is no array index that is assigned for the feature f_1. Similarly, we define ${f_M}^{-1} : \mathbb{Z} \rightarrow \mathit{Fea}(M)$ as a function that maps a given array index to the feature it represents. Thus, gene crossover is just the array exchange at a certain index, and gene mutation is bit flipping of the value at a certain index of the array.

To serve as the goals of malware generation, we propose three objective functions in the evolution of malware: *aggressiveness*, *evasiveness* and *detectability*. As the results

of the arms race, malware are getting more aggressive with minimum evasion features needed, but less detectable [3].

Given a chromosome x, we represent it as a bit vector of attack and evasion features, where $\{f_1^a ... f_n^a\}$ denotes the set of n attack features and $\{f_1^e ... f_m^e\}$ denotes the set of m evasion features. The objective functions are defined as follows:

Definition 7. **Aggressiveness** means the severity of damages that malware may cause to users, which is measured by the number of contained AFs and formally defined as

$$\mathscr{F}_1(x) = \sum_{i=1}^{n} \| f_i^a \| \tag{4.1}$$

where $\| f_i^a \|$ returns 1 if f_i^a is selected and returns 0 if not.

Definition 8. **Evasiveness** means the efforts to hide the malicious intent and evade the detection. Attackers want to minimize such effects of using evasion techniques to evade detection. It is measured by the number of contained EFs and defined as follows

$$\mathscr{F}_2(x) = \sum_{i=1}^{m} \| f_i^e \| \tag{4.2}$$

where $\| f_i^e \|$ returns 1 if f_i^e is selected and returns 0 if not.

Given a chromosome x and the set of AMTs S_d, $\{d_1 ... d_t\}$, introduced in Section 2.3.2, we have the following definition for the last objective function:

Definition 9. **Detectability** means the difficulty in detecting the malware. It can be measured by detection results of AMTs and defined as follows

$$\mathscr{F}_3(x) = (\sum_{i=1}^{|S_d|} \mathscr{D}_i(x)) / \mid S_d \mid \tag{4.3}$$

where $\mathscr{D}_i(x)$ returns 1 if the malware of x is detected by the tool d_i and $\mid S_d \mid$ denotes the number of the tools.

4.4.1 Feature Selection via IBEA

Rather than encode three objectives into one weighted fitness function, we treat all the three objectives equally and solve *Multi-objective Optimization Problems (MOPs)* using the Pareto dominance relation [138].

A k-objective optimization problem could be written in the following form[2] (in our case, $k = 3$):

$$\textit{Minimize } \vec{\mathscr{F}} = (\mathscr{F}_1(x), \mathscr{F}_2(x), ..., \mathscr{F}_k(x)) \tag{4.4}$$

where $\vec{\mathscr{F}}$ is a k-dimensional objective vector and $\mathscr{F}_i(x)$ is the value of $\vec{\mathscr{F}}$ for ith objective.

Definition 10. Given two chromosomes $\vec{x}, \vec{y} \in B^n$ and an objective vector $\vec{\mathscr{F}} : B^n \rightarrow R^k$, $\vec{x}$ **dominates** $\vec{y}$ ($\vec{x} \prec \vec{y}$) if

$$\forall\, i \in \{1, ..., k\} \quad \mathscr{F}_i(\vec{x}) \leqslant \mathscr{F}_i(\vec{y}) \tag{4.5}$$

$$\exists j \in \{1, ..., k\} \quad \mathscr{F}_j(\vec{x}) < \mathscr{F}_j(\vec{y}) \tag{4.6}$$

otherwise $\vec{x} \nprec \vec{y}$

Definition 11. Given chromosomes $\vec{x}$ and a set of chromosomes $S_{\vec{x}}$, $\vec{x}$ is **non-dominated** iff

$$\forall \vec{x}_i \in S_{\vec{x}} \quad \vec{x}_i \nprec \vec{x} \tag{4.7}$$

Algorithm 3 shows how IBEA guides the feature selection. The input of this algorithm is the feature model of Android malware, the number of iterations for the generation process, and the population size. In the beginning, it creates the initial population, randomly selecting features in the feature model (line 4 to 6), otherwise we can generate new malware derived from the existing malware (line 8 to 14). First, it selects two candidates with a probability and do an *ibea_crossover* operation (line 10) to generate a

[2]Evasiveness and detectability need to be minimized, but aggressiveness needs to be maximized. In implementation, maximizing is minimizing the negative value of the objective.

Algorithm 3: Multi-objective guided malware generation

Input: $featureModel$: the feature model of Android malware, $maxIter$: the maximum number of iterations of the generation process, $popSize$: the size for each generation

Output: $allMal$: a list of malicious apps with different feature combinations

```
1  allMal ← ∅
2  for 1 to maxIter do
3      newGeneration ← ∅
4      if allMal == ∅ then for 1 to popSize do
5          malware ← randomFeatureSelection(featureModel)
6          newGeneration ← newGeneration ∪ {malware}
7
8      else for newGeneration.size() < popSize do
9          select i, j ∈ [1, |allMal|]
10         malware = ibea_crossover(allMal[i], allMal[j])
11         newGeneration ← newGeneration ∪ {malware}
12         select k ∈ [1, |allMal|]
13         malware = ibea_mutation(allMal[k])
14         newGeneration ← newGeneration ∪ {malware}
15
16     for mal ∈ newGeneration do
17         evaluate(mal)
18     for mal_i ∈ newGeneration do
19         if ∃ mal_j | mal_j ∈ newGeneration ∧ i ≠ j • mal_j ≺ mal_i then
             newGeneration ← newGeneration \ mal_i
20
21     if newGeneration ⊂ allMal then break
22
23     allMal ← allMal ∪ newGeneration
24 return allMal;
```

new malicious app. Second, it selects one candidate in a probability and do an *ibea_-mutation* operation (line 13) to generate a new malicious app. Once the generation is created, all apps in this generation are evaluated by the proposed three objectives to get the fitness value (line 16, 17). The algorithm utilizes the Pareto dominance relation to remove malware that is dominated by others in the evaluation (line 18 to 19). The algorithm will stop once the selected features converges (line 21) or exceeds the maximal times of iteration (line 2).

4.4.2 Construction of Malicious Behaviors

After selecting features for a new malicous app from the feature model, we introduce an intermediate language, called *Behavior Description Language (BDL)*, to model concrete malicious behaviors with implementation details based on the selected features. The step is to guarantee the consistence and correctness between the feature model and the concrete implementation derived from it.

Behavior Description Language. The selection of features by the evolutionary algorithm just decides the malicious intent at the requirement level. To seamlessly glue the corresponding code of these selected features and attain a workable app, we introduce BDL to make an initial instantiation of these features, and bridge the gap between the feature model and the final implementation. We design the syntax of BDL inspired by Feng *et al.*'s work [42]. We add the implementation details (e.g., context [44]) into BDL to make it close to the code implementation, in a model-driven way of code generation.

Backus Naur Form of BDL. We present the partial BNF of BDL in Fig. 4.4 (refer to [133] for the complete definition of BDL). One app contains one or more flows, i.e., $\langle APP \rangle := \langle FLOW \rangle^+$. In attack of privacy leakage, one malicious flow is a concrete malicious behavior. It defines the concrete operations executed in apps. $\langle FUNCTION \rangle$ is the building block for a malicious flow, and it denotes how an app operates during the flow from the source to the sink at the atomic level. One function consists of three elements — $\langle COMPONENT \rangle$, $\langle POINTCUT \rangle$ and $\langle OPERATION \rangle$, where $\langle COMPONENT \rangle$ denotes the component, the building blocks of Android apps, $\langle POINTCUT \rangle$ denotes the methods where malicious behaviors are located, and $\langle OPERATION \rangle$ denotes the operation of malicious behaviors.

An Illustrative Example. For example, IBEA in step 1 selects android.provider.Telephony.SMS_RECEIVED as the concrete trigger feature, SMS::INCOMING_SMS as the concrete source feature, HTTP::APACHE_POST as the concrete sink feature.

Intuitively, the selected AFs access incoming SMS messages, store them into a local variable, and finally send the information out by posting a message via *Apache HTTP library*. The permission feature android.permission.READ_SMS is required.

$$
\begin{aligned}
&\langle APP\rangle && ::= \langle FLOW\rangle^{+}\\
&\langle FLOW\rangle && ::= \langle FUNCTION\rangle('\rightarrow' \langle FUNCTION\rangle)^{*}\\
&\langle FUNCTION\rangle && ::= \langle COMPONENT\rangle' ::' \langle POINTCUT\rangle' ::' \langle OPERATION\rangle\\
&\langle COMPONENT\rangle && ::=' ACTIVITY' |' SERVICE' |' BROADCAST\_RECEIVER'...\\
&\langle POINTCUT\rangle && ::=' POINTCUT\_ONCREATE' |' POINTCUT\_ONSTART'...\\
&\langle OPERATION\rangle && ::= \langle SOURCE\_SIG\rangle \mid \langle STORE\_SIG\rangle \mid \langle FETCH\_SIG\rangle...
\end{aligned}
$$

Figure 4.4: Parts of BNF for BDL

MYSTIQUE first constructs meaningful and valid behaviors based on the selected features. Different from the abstract feature model, we need to consider both the context of specific features and the dependencies in between. An app needs to register an instance of class BroadcastReceiver that listens the event of android.provider.Telephony.SMS_RECEIVED. Moreover, the acquisition of the incoming SMS message needs to be done in the context of method onReceive. Last, the operation of sending out information is also carried on in method onReceive. The BDL for this example is as follows:

```
BROADCAST_RECEIVER::POINTCUT_ONRECEIVE::SOURCE(SMS::
INCOMING_SMS)→ BROADCAST_RECEIVE::POINTCUT_ONRECEIVE::
SINK(LOCAL_VARIABLE, HTTP::APACHE_POST)
```

4.4.3 Code Assembly

In this section, we elaborate how to assemble code according to the constructed malicious behavior in BDL. The input of this step is a list of malicious behaviors, in the form of BDL. The output of this step is the assembled source code of the malware.

There are two traverse iteration processes in this step. First, we traverse the existing flows in BDL, then traverse functions in each flow. As described in Section 4.4.2, one function consists of three elements, i.e., component, pointcut and operation. It checks whether the component exists or not. It would create a new component if not, and add this component into the app. The corresponding operation is instrumented into the specific pointcut (i.e., method), and then the pointcut is instrumented into the very component. Meanwhile, in this step, we also write scripts to automatically check

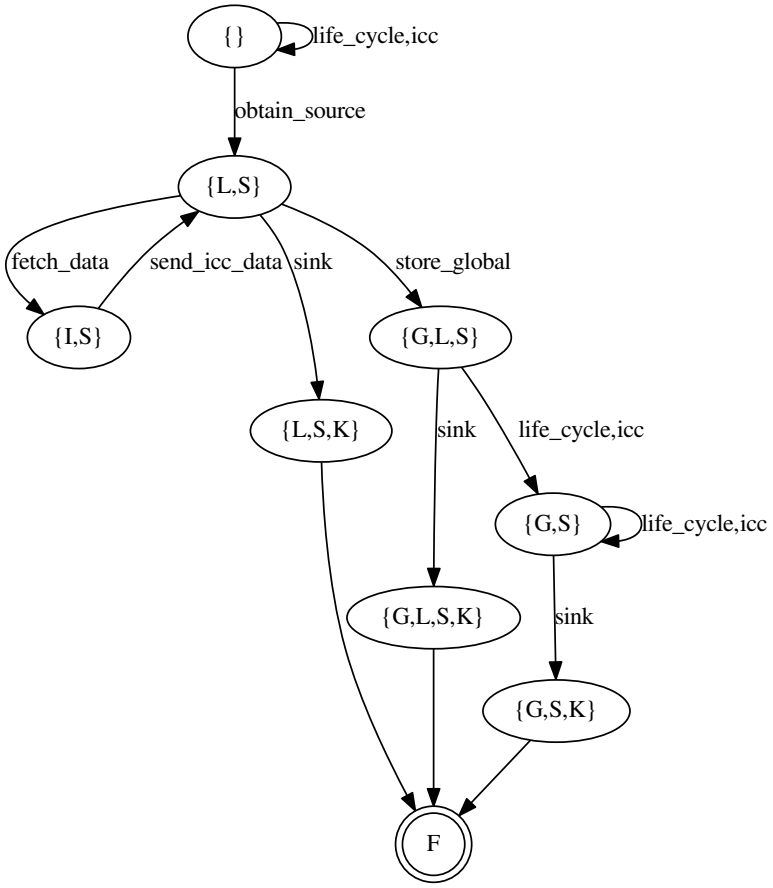

FIGURE 4.5: State diagram of flows

constraints from feature dependencies and context, and generate the configuration files, such as *AndroidManifest.xml*.

Constraints in Assembly. To assure the soundness and validity of the assembled code, we have the following rules:

- **Feature dependency constraint**. Feature model has features dependencies inside, which are constraints to be satisfied in code assembly. For example, permission feature android.permission.READ_SMS is needed for the BDL example in Section 4.4.2; the permission feature android.permission.READ_PHONE_STATE is needed for feature getDeviceId. When a component attempts to start an activity via ICC, in order to maintain the activity list in the history stack, it has to set the flag FLAG_ACTIVITY_NEW_TASK.

- **Context constraint**. BDL model embeds extra constraints to be verified in code assembly, specifically, some operations can only occur in a specific context. For

example, the incoming SMS message is only accessed in the context of onReceive, and the receipt of ICC in a service is onStartCommand. All these constraints are not feature-relevant at requirement level, but specific to implementation. All these constraints should be satisfied to avoid runtime exception.

4.4.4 Evasion Application

For information leakage, AFs and EFs are orthogonally separated, which implies the independence between the choice of AFs and EFs. After AFs and EFs are selected by IBEA, the EFs are categorized into two types: flow based ones or transformation based ones.

Evasion based on source-sink flow. This evasion is applied in constructing BDL models, and the purpose is to complicate the flow between the source and the sink. One malicious behavior may stretch through multiple components. To ensure a valid obfuscated flow for malicious behaviors, we identify a state diagram for state transition in Fig. 4.5, of which states are presented with five binary flags: **L**, whether the current context has the local variable of sensitive information; **G** whether the current context has the global variable of sensitive information (§ 4.3.2); **I**, whether the current context has the sensitive information received from ICC; **S**, whether the current context has carried on the *source* operation; **K**, whether the current context has carried on the *sink* operation. Note that we consider one malicious behavior is completed once it carries on a sink operation.

Evasion based on transformation. This evasion is applied after the step of code assembly. DROIDCHAMELEON can directly work with the deployment package of Android app. For the 12 transformations mentioned in Section 4.3.2, we provide 12 EFs. If any EFs are selected by IBEA in step 1, we will later apply the corresponding transformations.

4.4.5 Objective Evaluation

According to Definition 7-9, we calculate the fitness value for each generated malicious app. The fitness value is used as the guidance to the feature selection in the next generation. We inspect all apps in the new generation. If no new feature combination is produced, the evolution process converges and would be terminated as line 21 in Algorithm 3. Finally, we get the collection of new generated malware that serves the benchmark for auditing AMTs.

4.5 Malware and AMT Evaluation

MYSTIQUE is implemented in about 12K lines of Java code. Moreover, test scripts written for experiments are of 1K lines of Shell and Python. All the experiments are conducted on a Ubuntu 14.04 machine with Intel Xeon(R) CPU E5-16500 and 16G memory.

4.5.1 Hypothesis of Anti-malware Tools & Research Questions

According to the general analysis techniques employed in detection, AMTs can be categorized into five types (see Section 2.3.1). In this study, we focus on evaluation of different detection mechanisms rather than comparison of the particular tools.

We select as assessment subject 9 tools—DREBIN, ADAGIO, ALLIX, REVEALDROID, SCANDROID, FLOWDROID, ICCTA, DROIDSAFE, TAINTDROID and 57 anti-virus software in Table 2.1 that cover all types of evidence and knowledge base. Each tool checks several types of evidence. Evidence like UP, HC, CV, DF and AB are commonly considered in all tools. Note that for anti-virus tools, we leave question marks for the evidence they use, which is unknown to research community. However, they often rely on signatures for detection.

Before auditing AMTs, We propose the following four commonsense hypothesis:

Hypothesis 1. Mainstream AV tools, which rely on signature or pattern based approaches, cannot detect the variants of the existing malware, even those with similar attack features.

Hypothesis 2. Evasion features in privacy leakage, e.g., flow complication, can help the malware to evade the detection, despite which detection approach is used by the AMTs.

Hypothesis 3. AMTs based on dynamic analysis should be more accurate than those based on static analysis or machine learning, regardless of the time and the difficulty in setup.

Hypothesis 4. The human check of malware in online app store, involving both static and dynamic analysis, is the most complete and sound solution to detect malware in reality.

In this section, our experiments are aimed to answer the following research questions:

1. **RQ 1.** Are the modularized AFs and EFs valid? Is the generated malware valid and workable?
2. **RQ 2.** Can we use the generated malware to audit AMT? Are the Hypothesis 1 to 4 in Section 4.5.1 accepted or rejected?
3. **RQ 3.** Is our generated malware representative? How useful is MYSTIQUE in generating malware?

4.5.2 Evaluation Subjects

To evaluate the effectiveness of MYSTIQUE and the defense capabilities of AMTs, we generate multiple sets of malicious apps for different evaluation targets with MYSTIQUE. The malware is grouped based on its attack targets, and covers multiple attack and evasion features. On the other hand, we use the malware to test the defense capabilities of AMTs, especially, the state-of-the-art public AMTs introduced in Section 4.5.1. The evaluation subjects are described in the following two aspects.

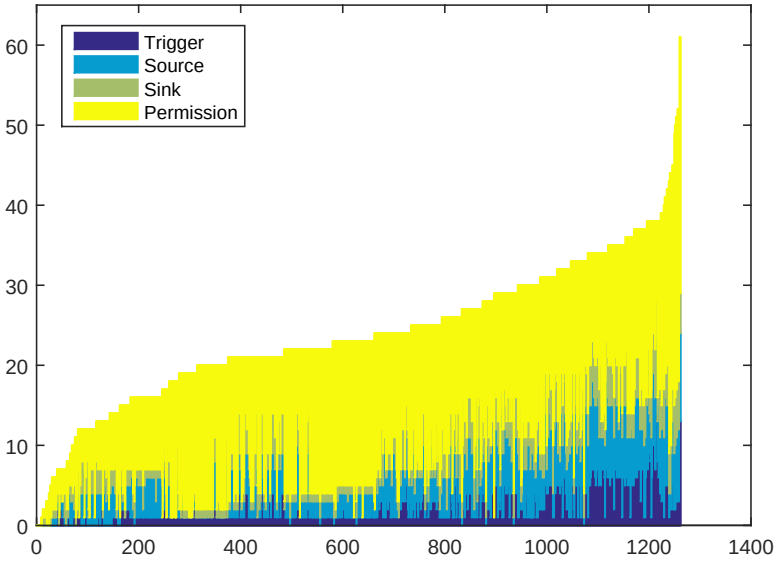

FIGURE 4.6: Cumulative AFs in GENOME samples

Offence: to evaluate the strength of the malware generated using MYSTIQUE. Each malware sample has at least one attack target, which is listed in Section 4.3.1. We give feature labels for malware to assess the attack capabilities. All the features used in MYSTIQUE feature model are manually summarized from the 1,260 malware samples in GENOME. Totally, we have 266 attack features and 14 evasion features in our feature model (§ 4.3.1). We sketch a diagram in Fig. 4.6 of the cumulative distribution for each kind of AF defined in GENOME. Since EFs are difficult to be categorized from the code, we do not show the distribution of EFs.

Defense:. to evaluate the four types of tools (§ 4.5.1) to cover a complete protection from three aspects: *untrusted app analysis*, *install-time checking*, and *continuous run-time monitoring* [18]. We need an initialization for machine learning and dynamic analysis tools. For machine learning tools, we select all 1,260 malware samples in GENOME, and 1,260 benign apps from Google Play as their training set. For dynamic analysis tools, we implement a driver in Python to simulate all possible triggers in our scope, e.g., starting an app, receiving an SMS message, changing the geography location. Interested readers can refer to the trigger list in [133].

Type	Feature	DR (%)
Source	TELEPHONY::SIM_SERIAL	21.1
Source	TELEPHONY::SIM_COUNTRY	12.5
Trigger	BROADCAST::android.bluetooth.device.action.NAME_CHANGED	12.5
Trigger	BROADCAST::android.intent.action.ACTION_SHUTDOWN	12.5
Trigger	BROADCAST::android.intent.action.PACKAGE_REMOVED	12.5
Source	SMS::INCOMING_SMS	1.6
Source	BUILD::SDK_INT	1.6
Trigger	BROADCAST::android.provider.Telephony.SMS_RECEIVED	1.6
Trigger	BROADCAST::android.intent.action.PACKAGE_RESTARTED	1.6
Sink	HTTP::SOCKET_GET	1.6

TABLE 4.2: The significance of attack features in detection

4.5.3 RQ1: Validity of Generated Malware

We validate the generated malware from three aspects.

Validity of Single Feature. For each of 266 AFs, we handcraft a blank Android program and generate the malware with the single AF. Then we execute the malware to verify whether it can successfully steal the information. For each of 14 EFs, we also separately apply it to complicate a basic information flow between a source and a sink. In the experiments, we finally assure that each AF can leak information and each EF can complicate the information flow.

Proof of Program Synthesis. We assure that the flows of privacy leakage in malware are logically true. In detail, we verify the three phases of the automated malware generation: **p1**, feature selection (§ 4.4.1), **p2**, transformation from features to BDL model (§ 4.4.2), and **p3**, transformation from BDL model to code (§ 4.4.3).

- **Proof of p1**. In the process of feature selection, we select appropriate candidate features, conforming to the constraints in the feature model. It guarantees there are sufficient and necessary features to construct malware.

- **Proof of p2**. In privacy leakage, BDL models define trigger features and behavior features, while the corresponding required permission features are missing.
- **Proof of p3**. Constraints on the unique runtime environment of Android (§ 4.4.3) should be satisfied. For example, consuming operations in Android apps cannot be executed in the main thread, and hence we have to create a child thread to execute consuming operations. The BDL makes abstract presentation of flows valid in a real app. In code assembly, we write scripts to make sure all the implementation constraints are satisfied.

To sum up, for given features, this step is to assure that all the requirement and implementation constraints are satisfied.

Malware App Validation. The last step is to valid the final malware app to test whether it can leak privacy information. To this end, we set the target URL and phone number to our *honeypot* that the information would be sent to. We use the running example to illustrative the validation process. We generate a malicious app using the features of malware in the running example (Fig. 4.1). The selected features are as follows:

Triggers-MAIN::STARTUP

Sources-TELEPHONY::IMEI, TELEPHONEY::PHONE_NUMBER

Sinks-HTTP::APACHE_POST

(dependencies) PERMISSION::READ_PHONE_STATE

MYSTIQUE constructs one flow of privacy leakage based on selected features, which is presented below in the form of BDL.

```
ACTIVITY::POINTCUT_ONCREATE::SOURCE(TELEPHONY::IMEI,
TELEPHONY::PHONE_NUMBER)→ACTIVITY::POINTCUT_ONCREATE::
SINK(LOCAL,HTTP::APACHE_POST)
```

We set the target URL to our *honeypot* web site, in which there is a responding web page written in PHP to store the received message from the generated malware. Since

there are 30 types of sources in the feature model, we use MYSTIQUE to generate 30 malicious apps accordingly, each of which contains one kind of sources. For simplicity, we construct one flow that satisfies the constraints defined in the feature model for the privacy leakage, by selecting one satisfiable trigger and sink, and setting up the acquired permissions. We execute them on a physical Android device. Our honeypot successfully collects all sensitive information sent by these malicious apps.

4.5.4 RQ2: Auditing of AMTs

In this section, we aim to evaluate the AMT using the generated malware and test the four hypothesis.

First, we test the deployed AMTs on GENOME malware as the baseline understanding of AMTs. Note that we only choose malware with privacy leakage attack, which contains 78% of the 1,260 samples in GENOME. The results are presented in Table 4.3, where machine learning tools and anti-virus tools perform well in detecting existing malware. As the dataset GENOME originated from 2010, anti-virus tools (AVTs), which are mainly based on signature and pattern matching, can accurately detect the malware with a recall of 71.9% on average. There are still some AVTs that perform poorly, e.g., Bkav (0%), CMC (0%), Malwarebytes (0%) and TheHacker (0%). Since machine learning tools use 60% of malware samples in GENOME as the training set and the remaining 40% as the testing set, they outperform the other tools with a higher recall.

Static analysis and dynamic analysis are more time-consuming compared to the previous two approaches, due to the program analysis they conduct. Static analysis tools has yet achieved around 48.4% of detection ratio of GENOME malware. For the dynamic tool TAINTDROID, it fails to detect existing malware in GENOME. The problem is attributed to the limited support of TAINTDROID to source or sink types, and the compatibility issues when running out-of-date malware in latest Android OS.

Second, we use MYSTIQUE to generate 100 generations of malware without evasion features to evaluate the detection ratio (DR) of AMTs. Then we add evasion features into the malicious apps to re-evaluate the DR. As shown in Table 4.4, there are two

TABLE 4.3: Detection ratio of privacy leakage malware in GENOME

Malware Family	# Samples	Detection Ratio (%)			
		DA	SA	ML	AV
DroidKungFu3	309	0	53.7	100	70.2
AnserverBot	187	0	51.0	99.7	74.7
BaseBridge	122	0	60.7	99.2	73.0
DroidKungFu4	96	0	10.2	100	70.3
Geinimi	69	0	40.1	99.3	71.9
Pjapps	58	0	45.6	98.9	71.6
KMin	52	0	37.8	100	72.0
GoldDream	47	0	55.6	100	70.4
DroidKungFu1	34	0	60.2	100	75.6
DroidKungFu2	30	0	67.0	100	73.7

TABLE 4.4: The objective value of generated malware during evolution

Gen	#Vars	AFs				#EFs	Detection Ratio (%)							
		#Triggers	#Sources	#Sinks	#Perms		DA	DA(E)	SA	SA(E)	ML	ML(E)	AV	AV(E)
10	50	35.9	13.4	4.6	74.9	5.6	17.2	13.4	32.5	12.5	42.5	41.7	0.0	0.0
20	50	31.8	7.6	4.5	82.6	7.5	34.2	14.3	25.0	13.5	25.0	22.5	0.0	0.0
30	50	33.8	9.8	3.1	75.2	4.8	24.5	14.3	27.5	15.0	20.0	20.0	0.0	0.0
40	50	29.5	9.9	2.4	78.3	5.1	14.1	14.1	17.5	15.9	32.5	29.5	0.0	0.0
50	50	32.9	8.2	2.4	81.9	8.0	22.0	15.9	17.5	0.0	27.5	25.0	0.0	0.0
60	50	31.2	10.5	2.2	80.5	7.5	21.0	10.5	17.5	12.8	27.5	22.5	0.0	0.0
70	50	27.6	10.5	2.4	74.5	6.2	6.7	4.8	17.5	15.0	25.0	22.5	0.0	0.0
80	50	28.5	10.3	4.8	70.3	3.1	19.5	14.2	18.2	12.5	27.3	25.0	0.0	0.0
90	50	33.3	9.0	2.9	77.5	6.6	5.6	5.6	10.0	5.8	25.0	21.0	0.0	0.0
100	50	36.9	10.0	4.0	76.5	5.4	10.2	8.7	10.0	6.0	22.3	20.0	0.0	0.0

columns for each kind of AMTs, of which the first column is the DR **without** evasion features, and the second column "(E)" is the DR **with** evasion features. All the values of DRs are calculated as the average values amongst tools of a specific type. We summarize the hypothesis testing results as follows.

H1. The Susceptibility of AVs to Unknown Malware. Mainstream AVs employ

signature- or feature-based approaches. The detection capabilities depend on the completeness and timeliness of malware database, and also the abstraction of malware. Generally, they perform very well in detecting known malware as in GENOME experiment above: they achieve a 71.9% recall on average, and 27 (out of 57) AVs can even detect at least 99% of malware samples in the experiment. However, they perform poorly in detecting our generated 0day malware. According to the detection results of our generated malware, only 18 generated malware samples can be detected by the union of these AVs. For example, ESET-NOD32 detects 3 malware samples as "a variant of Android/TrojanSMS.Agent.BLY". By further inspection, we find that the 3 samples steal the SMS messages. Specifically, they share one common behavior as below. It monitors the change of the Content Provider of SMS, steals all SMS messages, and sends out to a specific remote server.

```
CONTENT_OBSERVER::POINTCUT_ONCHANGE::SOURCE(SMS::ALL)→
CONTENT_OBSERVER::POINTCUT_ONCHANGE::SINK(SMS::SEND_MESSAGE)
```

We can conclude that AVs have made efforts to infer the semantics of code as the behavior is split into two methods. However, the inference is quite limited. We crafted malware samples by employing evasion techniques, which cannot be detected any more. In general, we consider H1 is **accepted**.

H2. The Insignificant Impact of Evasion Techniques. We have generated two malware datasets, one of which contains malware samples without any evasion features, and the other contains malware samples with arbitrary evasion features. From the comparison of detection results, evasion features rarely effect the detection results of AVs. It can help to evade the detection of dynamic and static analysis (43.7% of reduction in DR). Since the dynamic analysis tool TAINTDROID tracks the flow of information in the system, it fails to detect the privacy leakage once the flow is complicated by involving ICC or implicit data flow. The static analysis tools that perform a code analysis from the source to sink, can overcome complicated transformation attacks and behavior-level evasion techniques. For example, ICCTA takes into the account ICCs during different components of apps, can identify behaviors of privacy leakage occurring across multiple components. However, static analysis in ICCTA still has some flaws. It cannot track

the data flow across persistent storage, such as file, SQLite or shared preferences. Static analysis tools usually employ API-matching to identify sources and sinks. Therefore, they can be easily defeated by involving dynamic loading techniques, such as reflection, constant encryption. Moreover, for machine learning based tools, evasion features have a little impact on DR, which is not significant enough (the differences of ML and ML(E) in Table 4.4 are within 5%). We observe that the higher #EFs does not necessarily lead to a lower DR.

Thus, we consider that H2 is **partially accepted** — certain evasion can only work for certain detection approaches and too many evasions may not better bypass the detection.

H3. Diverse Detection Capabilities of AMTs. Based on the detection results to our malware benchmark, we test H3 by evaluating the weakness and strength of each type of approaches.

- Dynamic analysis is a kind of black box testing, which focuses on the input and output of sensitive information to apps, while they do not consider how the behavior is implemented. Therefore, the detection capabilities depend on the coverage of sources, sinks and the communication channels between. Our experiments show that TAINTDROID can track sensitive information obtained from specific Android APIs, such as getDeviceId and getLine1Number. It does not track the information from incoming SMS message and Content Provider, etc. It performs well for the communication channel ICC and file-based channel. However, SQLite and shared preferences can help bypass its detection.

- Static analysis is more scalable than dynamic analysis. However, it lacks of information during runtime and thereby its capabilities are limited. Nowadays, there are some works [139] using *symbolic execution* to mitigate the lacking of runtime information.

- We compare the detection results of two malware sets, one of which has more attack features and the other has less attack features. The dataset with more attack features is more likely to be detected, while machine learning based approaches are susceptible to malware with less attack features. Another comparison occur

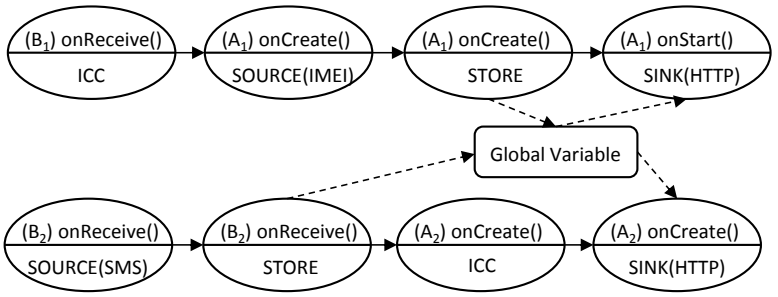

FIGURE 4.7: Malicious behaviors to be repackaged

between two tools REVEALDROID and DREBIN. Although DREBIN has considered more features, its detection ratio is improved a lot. Therefore, the significance, rather than the number, of features can better facilitate the detection. As shown in Fig. 4.2, we list five attack features which are easiest to be detected, and five attack features which are hardest to be detected (refer to [133] for a complete list of significance of attack features in the feature model for details).

- It is reasonable for AVs to use a fast approach with a low false positive rate. Our observation is that AVs mainly aim at detecting known malware. Hence, AVs work in a reactive way, not in a proactive way.

To sum up, we consider that H3 should be **rejected**. Considering the detection results of TAINTDROID in Table 4.4, we cannot confirm that dynamic tools can produce high detection accuracy, although they can provide more accurate information in detection. The problem lies in the difficulty in triggering malicious behaviors in execution. Note that due to the unavailability of other dynamic tools, we cannot generalize our conclusion for all dynamic tools.

TABLE 4.5: The capabilities of vetting process in modern marketplaces

#Benign Base	Google Play	GetJar	SlideMe	TorrApk
1	✓	✗	✓	✗
2	✗	✗	✗	✗
3	✗	✗	✗	✗

H4. Strong Vetting Process in Modern App Stores. Modern Android app stores employ multiple techniques to inspect the submitted apps and protect their marketplaces. Google Play has turned from an offline dynamic analysis-*Bouncer* [140] to a manual check by human experts [141]. Currently, Android app stores *GetJar* [118], *SlideMe* [122] and *TorrApk* [142] all inspect the submitted apps by human experts.

Since our generated malware has no normal functionalities other than malicious behaviors, it got rejected when we submit it into these four app stores. To address this, we download three open-source benignware, which have been verified by AMTs and approved by Google Play. We inject our malicious behaviors into their source code, repackage them and then submit them to the four Android app stores. One example of malicious behaviors is shown in Fig. 4.7. And it acquires 5 permissions and steals SMS messages and identity information of device into a particular server.

For each of the 3 benignware (benign base), we select 4 malware samples from our benchmark and inject them into the benign base. Here are the 4 malware samples: 1) one malicious app without evasion features; 2) one malicious app is generated by adding evasion features into the first app; 3) an optimal malware sample in our benchmark. 4) a random choosen one from our malware benchmark. The 4 different malicious apps from the same benign base is submitted to the four different app stores. Table 4.5 shows the detection ratio of these apps by AMTs, ✗ and ✓ indicate an app is approved (not detected) and rejected (detected) by the corresponding app stores, respectively. From this experiment, we can conclude that the vetting process of Android app stores still have severe flaws, and can be easily bypassed. Although human inspection can judge the quality of apps of high confidence, the security of apps is not fully inspected.

According to our observations, we consider that H4 should be **rejected**. Note that the malware samples that are used for injection are with a ratio of 0% to 22.8% to be detected. Thus, the vetting results are not significantly better than the results of our AMTs. We suspect that the vetting process also uses the AMTs for detection.

4.5.5 RQ3: Representative Malware and Usefulness of Mystique

This section first explains the usefulness of MYSTIQUE in generating malware and evaluating anti-malware. Then, we show how representative the malware generated via evolution mechanism is.

4.5.5.1 The Usefulness of MYSTIQUE

FODA of Android malware (§4.3.1) helps attain the precious domain knowledge on Android attack behaviors. In MYSTIQUE, we maintain the traceability of all the features and their corresponding code. Hence, for each malware generated by MYSTIQUE, all selected features (e.g., triggers, source, and sink) are labeled. These labels can provide a good indexing mechanism for the collection of generated malware, which can facilitate the malware management.

In addition to using IBEA to generate the malware, MYSTIQUE also supports the customized malware generation — the user can decide which features are selected for malware generation. MYSTIQUE verifies the feature constraints, and generates the malware if no feature conflicts are found. The usefulness and flexibility of MYSTIQUE benefit from the SPLE architecture. Besides, rather than randomly mutate the malware, MYSTIQUE produces more aggressive yet less detectable malware via evolution process. We explain this in Section 4.5.5.2.

4.5.5.2 The Rapid Acquisition of Optimal Malware

We conduct a controlled experiment to assess the effectiveness of MYSTIQUE to obtain optimal malware from the attacker's view. The basic idea is to use a small set of AFs and EFs for fast convergence, and evaluate the malware when the evolution stops.

The experiment is conducted as follows: 1) pick up 10 malware samples which can be detected. 2) use IBEA to generate new variants by combining or mutating features in the initial population of malware. 3) stop if no more optimal malware is generated.

The 10 samples are from malware family DroidKungFu3, AnserverBot, BaseBridge, DroidKungFu4, Geinimi, Pjapp, KMin, GoldDream, DroidKungFu1 and DroidKungFu2 as shown in Table 4.3. The extracted features are as follows. In addition, we consider all 14 types of evasion features in this experiment.

Triggers:

[T_1] STARTUP,

[T_2] android.intent.action.BOOT_COMPLETED,

[T_3] android.intent.action.BATTERY_CHANGED,

[T_4] android.intent.action.NEW_OUTGOING_CALL

[T_5] android.provider.Telephony.SMS_RECEIVED,

Source:

[SU_1] PACKAGE::INSTALLED_APK,

[SU_2] SMS::ALL,

[SU_3] SMS::INCOMING_SMS,

[SU_4] TELEPHONY::IMEI,

[SU_5] TELEPHONY::IMSI,

[SU_6] TELEPHONY::PHONE_NUMBER,

[SU_7] TELEPHONY::SIM_SERIAL

Sinks:

[SI_1] HTTP::APACHE_GET,

[SI_2] HTTP::APACHE_POST,

[SI_3] HTTP::SOCKET_POST,

[SI_4] SMS::SEND_TEXT_MESSAGE

Permissions:

[P_1] android.permission.INTERNET

[P_2] android.permission.PROCESS_OUTGOING_CALLS

[P_3] android.permission.RECEIVE_BOOT_COMPLETED

[P_4] android.permission.READ_PHONE_STATE

[P_5] android.permission.RECEIVE_SMS

[P_6] android.permission.SEND_SMS

Evasion:

[E_1] Control based evasion

[E_2] Data based evasion

[E_3] Transformation attacks (12 types of transformation)

Initially, MYSTIQUE selects features randomly to construct 10 malware samples as the initial population. MYSTIQUE evolves based on the fitness value of newly generated malware. After 30 iterations, MYSTIQUE obtains the optimal malware of which the fitness values reach optimum in three objectives. The optimal malware contains 16 attack features and 3 evasion features. AFs in the optimal malware are { T_1, T_3, T_5, SU_1, SU_2, SU_4, SU_5, SU_7, SI_1, SI_2, SI_3, SI_4, P_1, P_2, P_3, P_6}, and EFs contains control

based evasion, data based evasion and one transformation. We put the optimal malware and more details on our tool website [133] for public observation.

4.6 Discussion

4.6.1 Threats to Validity

The internal threats for experiment results are from three aspects. First, we mainly consider privacy leakage attack and their behaviors. However, the logic of this attack is quite straightforward and we have the limited number of AFs. Second, for EFs, we mainly take into account the flow complication and transformation attacks. We discuss the effects of obfuscation in Section 4.6.4. Last, we use default parameters and set-up of IBEA for malware evolution. Further investigation should be conducted to see the effects of different parameters and set-up.

The external threats mainly stem from the choice of malware samples for FODA. Currently, our malware samples for FODA are only from GENOME. As GENOME originated from 2010, it may contain lots of out-of-date malware. To ensure the timeliness of malware, we need to further investigate what malicious behaviors are contained in other malware collections. Another threat is due to the public availability of the AMTs that we can audit. Especially, for tools based on dynamic analysis, the source code is required for better debugging and testing the malware. At this moment, we only have the source code of TAINTDROID. In future, we plan to audit more AMTs based on dynamic analysis.

4.6.2 Extensibility of Mystique

MYSTIQUE can be enhanced in future from the following aspects:

- The feature model is extensible for including more kinds of features (both attack and evasion) to express various attacks in Android. As mentioned in Section 2.2,

feature model can cover other attacks similar to information leakage. For the collusion attack [130] and attacks using concurrency bugs, the feature model needs some extension to annotate the timed and shared information among collusive apps.

- The corresponding implementation of features can be easily extended to support more advanced mechanisms and programming languages. For example, one feature can be implemented by using Java reflection or C/C++ native code. The feature code modularization mainly requires the code can be orthogonally separated. In SPLE, the app generation can be done at the source code level, component level (component based SPL) or even service level (service oriented SPL).

4.6.3 Countermeasure for Generated Malware

According to our experiments on generated malicious apps and the detection results of AMTs, we present three suggestions for future research on Android malware detection.

- **A Refined Source&Sink Pattern**. Generally, the recognition of sources and sinks is the first step for static- and dynamic-analysis tools. Consequently, they need to track the flow of information (obtained by sources) in either the program or the runtime environment. However, most of existing works [22, 23, 25, 40] identify sources and sinks by doing a matching with Android APIs, such as SuSi [143]. For example, getDeviceId is recognized to obtain the IMEI code on device. However, there exist some sources which cannot be represented as APIs. For example, the number of incoming calls can be obtained from the context of ⟨PhoneStateListener⟩onCallStateChanged. Although SuSi includes the methods getLatitude and getLongitude as sources, malware can use the method toString instead to fetch the specific latitude and longitude to bypass AMTs' tracking, and **these kinds of sources exist in our benchmark**. Hence, one refined pattern for sources and sinks facilitates the detection of privacy leakage.

- **Full Consideration of Communication Channels**. There exist many communication channels in Android, through which information is transmitted. Besides

ICC provided by Android, malicious apps can communicate via system memory or persistent storage. In addition, there emerge side channel attacks in Android [144, 145]. All of these advanced techniques hinder the detection of malware. Therefore, modern detection approaches should follow the development of attacks firmly and supplement domain knowledge from time to time.

- **Correct Understanding of Malicious Behaviors**. Current approaches based on machine learning lack an understanding of essences of malicious behaviors. Features extracted from apps are usually separated or not directly relevant to malicious behaviors. Although machine learning tools achieve 91.4% on accuracy in the training, they can only detect less than 9.5% of generated malware in reality (§ 4.5.4). Therefore, with a tolerable loss of efficiency, machine learning based approaches can learn the essences of malicious behaviors deeply to increase their performance on Android apps in the wild. For example, they can employ static analysis to extract the relationship between different features [24, 41, 44].

4.6.4 Evasion vs. Obfuscation

In this thesis, we define and use evasion features rather than obfuscation features for malware generation. Software obfuscation is mainly used for hiding the program logic from manual check. Obfuscation could be used for property protection for benignware, yet it is also used by malware to evade detection that relies on signature or program analysis. Thus, from the view of defence, we consider evasion is evil while obfuscation is neutral.

In malware development, we define one of the attack objectives as the minimization of the number of used evasion features for escaping detection of AMTs (§4.4). However, from some attacker's view, he may also prefer more obfuscation — obfuscate the malicious code at bytecode level and fail the attempt of manual check. As our study is to audit AMTs, we do not cover how obfuscation can fail the manual check. Obfuscation is not useful for dynamic approaches, and it may even provide hints for machine learning based detection. A true case is that obfuscation on its own can be predictive features

to distinguish malicious and benign JavaScript [146]. Hence, in reality, too many obfuscations may not be helpful in evading detection of AMTs. In our study, we do not consider byte-code obfuscation.

4.7 Related Work

Android Malware Generation DROIDCHAMELEON [126, 129] integrates three types of transformation techniques and can generate obfuscated Android malware, which is used to evaluate state-of-the-art anti-virus tools. Aydogan and Sen [147] propose an approach to generate Android malware with a genetic algorithm. The newly generated malware originate from the crossover and mutation of malware in GENOME [5], and they conducted experiments to show that the new malware variants can easily bypass the detection of anti-virus tools. Cani *et al.* [148] employ μ GP to automatically create new malware which is undetectable for anti-virus tools, and injects malicious code into a benign app to construct a Trojan horse.

Malware Evasion Techniques. Christodorescu et al. [149] firstly give a formal definition for obfuscation, and these techniques can be used by hackers to modify their malware to evade the detection of anti-virus tools and analysis of security analysts. In order to hinder dynamic analysis of Android malware, Petsas et al. [150] propose three heuristics to check if malware is running on an emulated device or a real device, thereby decide whether to execute malicious behaviors. The three heuristics contain-static heuristics, such as IMEI code, routing table; dynamic heuristics, such as sensor data, and; hypervisor heuristics, such as QEMU scheduling. Maier et al. [151] construct *Divide-and-Conquer* attack to evade the detection of anti-virus tools. They design a tool SAND-FINGER to collect identifiers for each virtual environments of malware detection. They combine fingerprinting and dynamic code loading to transform the existing malware, moreover, use collusion attack to split one malicious behavior into multiple apps.

Anti-malware Auditing. VIRUSTOTAL [60] is a public web service which deploys over 50 commercial anti-virus tools, and generates an analysis report for Android apps.

AndroTotal [152] is a integrated framework to automatically test the detection capabilities of anti-virus tools. Christodorescu and Jha [149] leverage four types of obfuscation techniques to test the capabilities of commercial anti-virus tools. In addition, they propose an algorithm to extract the unique signature by which anti-virus tools use to identify malware. ADAM [153] is an automatic and extensible platform to test and audit Android anti-virus tools. It employs several transformation techniques to generate polymorphic malware, and test 10 prestigious anti-virus tools. DROIDCHAMELEON [126, 129] collects three types of transformation attacks in Android, and the authors have used these attacks to audit the off-the-shelf detection tools. Huang *et al.* [154] assess the detection capabilities of 30 top anti-virus tools from two aspects: malware scanning and engine updating. They reveal hazards of evasion in malware scanning, and null-protection windows during the update of engine.

4.8 Conclusion

We propose a feature model to describe the behaviors in malware for the ease of understanding and detection. We present MYSTIQUE, an Android malware generation framework to automatically generate malware with specific features. The generated malware is used to explore the agressivity of attack features, and efficiency of evasion techniques. We provide 10,000 generated malicious apps which can be used to evaluate the emerging AMTs and thereby help to enhance the security of Android ecosystem. In future, we will work out the feature model that covers all the other types of attacks in GENOME, and we will try to test more dynamic tools (if available) with more types of generated malware.

5

Evolving Android Malware with Dynamic Loading Technique

5.1 Introduction

According to a report from AV-TEST [155], the independent IT-security lab, 26 off-the-shelf anti-malware tools (AMTs) show high detection rate (DR) of above 90% for existing Android malware. This test report proves that the mainstream signature-based ATMs can effectively detect *existing* malware, provided with a comprehensive list of malware signatures. However, generally, the development of AMTs usually lags behind the advance of new attack or malware variants. The consequence of the arms race in Android security leads to the sophisticated malware, which may contain a variety of attack behaviors and evasion techniques (e.g., multiple-level obfuscation [126, 129],

new transformation attacks [126, 129] and collusion attacks [128, 130]). Besides, dynamically loaded malware is becoming increasingly severe. Existing benchmarks GENOME [5] and DREBIN [20] are not updated to the aforementioned attack or evasion features.

Several existing studies relate to Android malware generation. DROIDCHAMELEON [126, 129] integrates three types of transformation techniques to generate obfuscated malware, which are used to audit the AMTs. Aydogan and Sen [147] proposed to generate Android malware with a genetic algorithm. The newly generated malware came from the crossover and mutation of malware in GENOME [5], and they conducted experiments to show that the new malware variants can easily bypass the detection of AMTs. Cani *et al.* [148] uses μ GP to automatically create new malware undetectable for AMTs, and injects malicious code into benignware to create a Trojan horse.

To sum up, the aforementioned studies mainly adopt new evasion techniques or mutate malware samples for new possible variants. As shown in the study [148], using genetic programming (GP) to mutate malware faces one critical problem: deciding whether an evolved variant still retains the characteristics of malware is a major issue of the evaluator. Behavioral modification of existing malware via GP can neither guarantee the maliciousness of the generated one, nor produce malware with the desirable attack behaviors in a systematic way.

A desirable malware benchmark for AMT auditing should label each sample with the contained fine-grained *attack features*. We refer to *attack feature* (AF) as a step or a component (i.e., triggers, permissions or concrete behaviors) of a certain attack, which links to the configuration or implementation of the functional requirements (intention) of malware. For example, phishing malware usually contains three AFs: a faked GUI that tricks users to input the credentials, a source component to steal the credentials, and a sink component to leak the credential. Neither GENOME [5] nor DREBIN [20] explicitly labels the AFs inside each malware sample, not to mention allowing security analysts to derive new malware variants for auditing AMTs.

In our previous chapter, Android malware generation is treated as a software product line engineering (SPLE) problem [156], considering new malware variants as product

variants in software product line (SPL). We separate each common attack behavior into a basic reusable feature via domain analysis [157] — to modularize the AFs of malware (§ 5.3). In this way, we develop a meta-model (i.e., feature model in SPL, see § 5.2.1 and Fig. 5.2) of Android malware by modularizing AFs into *building blocks*. With SPL for malware generation, having a large set of valid and well-labeled malware is not challenging.

Our previous study shows that existing AMTs are susceptible to *new variants* of old GENOME malware [1]. The AMTs can detect 90% of GENOME malware on average. After we apply multi-objective evolutionary algorithm (MOEA) to combine different attack and evasion features that are modularized from GENOME, the DR sharply drops to 20-40% in 10 generations of evolution. Finally, for malware variants after 100 generations, the existing AMTs only detect 10-20% of them on average [1].

However, our tool MYSTIQUE used in previous study does not produce the attack of dynamically loaded malware [1]. Considering the severity of this attack [158], we want to audit whether the AMTs can detect the evolved malware that is assembled and loaded dynamically. Hence, in this study, we extend MYSTIQUE to be service-oriented and name it as MYSTIQUE-S. It adopts dynamic software product line (DSPL) techniques [159] and delivers the generated malware at runtime from the remote server to the client for evading detection.

Technically, MYSTIQUE-S consists of three major steps. First, its client app collects some hardware and software information on device, which is achieved by a simple scanning without root privilege. Then the information is sent to the server side of MYSTIQUE-S (§ 5.4). Next, the server automatically selects a set of AFs that satisfy the constraints on the user device, and generates the malicious code on the fly (§ 5.5). For example, the details of the user scenario (e.g., the model of device, OS version and installed AMTs) are analyzed and converted to constraints. To guide the AF selection, we propose three goals: *aggressiveness*, *latency*, and *detectability* (§ 5.5.2). Each AF has a score for latency and a score for detectability. Linear programming (LP) is applied to find the AFs that satisfy the constraints and optimize the three goals. Lastly, the malicious code is delivered to the client device via a web service, and executed

via the reflection mechanism (§ 5.6). We adopt the reflection mechanism offered by DEXCLASSLOADER [158], which can load *dex* files and execute the *class* files inside.

Different from the work in [132] which focuses on requirements in malware ontology analysis, we maintain the traceability between the AFs and their corresponding code. To assemble the code of different features, we introduce the behavior description language (BDL) (§ 5.6.1 and 5.6.2) to serve as the bridge between the high level AFs and the low level implementation code. Owing to the BDL representation, we validate and generate the malicious code in a model-driven way. Beyond our previous work [1], we also make the following novel contributions in this study:

- In previous study, only for the attack of privacy leakage, we build the malware meta-model and generate the variants [1]. Now, we complement the meta-model with more attacks like financial charge, phishing and extortion. We modularize the AFs of these attacks, and generate variants accordingly.
- MYSTIQUE-S adopts a service-oriented architecture to collect the client-end data and deliver the malware at runtime. Meanwhile, to support the model-driven malicious code generation, we propose the BDL to glue the high level features with their low level implementation code.
- Our work in [1] relies on MOEA, which is computationally costly. In this work, we adopt linear programming (LP) to select suitable attack features for optimizing the objectives of malware inventor, since LP can rapidly solve the constraints of feature model on the fly and avoid the evolution time of MOEA.
- Instead of using static detection or dynamic detection via virtual machine in the report [1], we evaluate our tool on 16 real Android devices. We observe that in most cases, the malicious code generated by MYSTIQUE-S are not detected. According to our finding, we propose some enhancements for the AMTs.

5.2 Background

5.2.1 Dynamic Software Product Line

SPLE is a software development paradigm that has received much attention in the last decade [157]. SPLE aims to reuse the commonality among the products inside the same family, and maintain the product variants in a systematic way. SPLE usually adopts the feature-oriented domain analysis (FODA) to identify the codebase and variant features [131]. The *codebase* refers to the same code shared by all the product variants, which is the implementation of the basic functionality of a software family (a set of similar products) [156]. *Variant features*, which are different extra functions, are used to satisfy the needs of various customers. Typically, SPLE includes two stages: *domain engineering* that builds the architecture consisting of the codebase and variant features, and *application engineering* that derives new products by applying variant features onto the codebase. Generally, automation of product derivation is the main advantage of SPLE.

Feature model (FM). The centric concept in SPLE is to extract the *feature model* [131] — a tree-like feature hierarchy that captures the structural and semantic relationships between features inside. The core task in application engineering is to choose a set of features to derive valid products that satisfy all the constraints [156] and optimize the product performance.

Given a feature f and its sub-features $\{f'_1, ..., f'_n\}$, there exist four types of tree-structure constraints (TCs) (see Fig. 5.2 for example). We list them and show their logical formula [160]:

- f'_i is a *mandatory* sub-feature — $f'_i \Leftrightarrow f$,
- f'_i is an *optional* sub-feature — $f'_i \Rightarrow f$,
- $\{f'_1, ..., f'_n\}$ is an *or* sub-feature group — $f'_1 \vee f'_2 \vee ... \vee f'_n \Leftrightarrow f$,
- $\{f'_1, ..., f'_n\}$ is an *alternative* sub-feature group — $(f'_1 \vee f'_2 \vee ... \vee f'_n \Leftrightarrow f) \wedge \bigwedge_{1 \leqslant i < j \leqslant n} (\neg(f'_i \wedge f'_j))$.

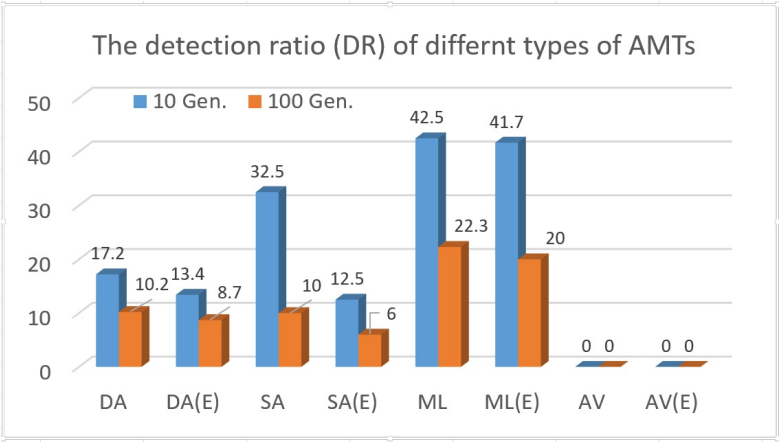

FIGURE 5.1: Results of AMTs auditing by using MYSTIQUE [1]

Further, given two features f_1 and f_2, three types of cross-tree constraints (CTCs) exist, i.e., *requires*, *excludes* and *iff* [160]:

- f_1 *requires* f_2 — $f_1 \Rightarrow f_2$,
- f_1 *excludes* f_2 — $\neg(f_1 \wedge f_2)$,
- f_1 *iff* f_2 — $f_1 \Leftrightarrow f_2$.

In traditional SPLs, variant features are bound to different products statically at compilation time (before the execution of the system). In contrast, adaptive systems support feature binding at runtime and are called dynamic SPLs (DSPLs) [159]. A recent progress in SPLE is the implementation of DSPL via the rapidly emerging paradigm of service-orientation (SO). By virtue of the dynamic composition of service, variants features can be loaded into the system dynamically according to user preferences and environmental scenarios. In SPLE, a feature model (e.g., that of Linux kernel) may contain thousands of features. It is a non-trivial problem to select an optimal set of features which satisfies the constraints (i.e., TCs and CTCs) among features. Selecting an optimal feature set represents a searching problem [161]. Such problem is normally addressed in SPLE community using techniques such as MOEAs.

5.2.2 Summary of Previous Study

In the previous study, we apply MOEA to mimic malware evolution [1]. In particular, two genetic operators are applied on the current generation to produce next malware generation: *gene crossover* (i.e., exchanging (attack or evasion) features of two samples) and *mutations* (i.e., mutating the selection of features of malware). To retain evasiveness and aggressiveness of malware in evolution, we define multiple evolution objectives (a.k.a. fitness functions) for selecting malware variants to survive into the next generation: 1) maximizing the number of attack behaviors, 2) minimizing evasion techniques needed and 3) minimizing the expected detection rate.

In Fig. 5.1, we summarize the results by two bars of each of four types of AMTs. The first one is the Detection Ratio for evolved malware without evasion features; the second one "(E)" is the DR for malware with evasion. "DA" denotes for dynamic based AMTs; "SA" for static based AMTs; "ML" for machine learning based AMTs; "AV" for the popular Anti-virus tools. After malware evolves from 10-*th* to 100-*th* generation, the DR of the audited AMTs sharply dropped. We attribute the low DR for evolved malware to the modularity offered by MYSTIQUE.

In this study, we extend [1] to support more types of attacks and the dynamic loading technique for advanced evasion [158]. To improve the efficiency, in MYSTIQUE-S, we adopt LP (not MOEA) to select AFs for malware generation.

5.3 Feature Model of Android Malware

To create new malware variants by reusing the attacks in existing malware, we first analyze the malicious code in malware benchmark GENOME [5] and recent malware samples. Then, we represent AFs as a feature model (FM) via FODA aided by the domain knowledge of security experts. In general, we categorize the AFs into three types, namely trigger, permission and behavior features in Section 5.3.1.

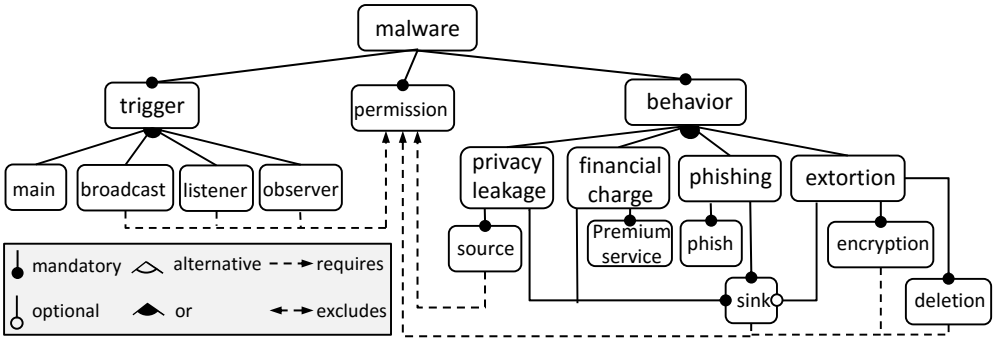

FIGURE 5.2: The partial feature model of Android malware

For the four types (excluding privilege escalation) of Android attacks introduced in Section 2.2, the corresponding feature model is partially shown in Fig. 5.2 under the *behavior* node.

Currently, we identify and modularize 92 attack features (§ 5.3.1), and extract the CTCs among these features. For the completeness of the classification, owing to the extendability of the feature model, we can always add new attack features into the feature model (e.g., privilege escalation [162]). Note that the feature model is a conceptual modeling of features, and we also keep the traceability between a feature and its modularized code in our built SPL (§ 5.3.2).

5.3.1 Attack Features

We identify different AFs according to the context, permission and functionality relevant to the attack, as shown below.

Trigger features refer to the configurations that customize the entry points for malicious attack behaviors. Triggers can be GUI-based or non GUI-based [44]. GUI-based triggers can be easily identified by end users or AMTs, since it requires interaction with visible GUI components [136]. In this thesis, we only consider non GUI-based triggers that have no interactions with users. Four types of triggers are identified in GENOME:

- Trigger feature main means that the related behavior can be triggered since the beginning of the lifecycle of an app;

TABLE 5.1: Parts of attack features in covered attacks

Source	**Category**
TELEPHONY	IMEI, IMSI, PHONE_NUMBER, etc
SMS	INBOX, INCOMING_SMS, etc
CALL	CALL_LOG, INCOMING_CALL, etc
BROWSER	BROWSER_HISTORY, etc
MEDIA	RECORD_AUDIO, etc
LOCATION	REAL_TIME_LOCATION, etc
BUILD	CODE_NAME, SDK, etc
CONTACT	CONTACT, etc
ACCOUNT	ACCOUNT, etc
STORAGE	EXTERNAL, etc
PACKAGE	INSTALLED_APK, etc
Sink	**Category**
HTTP	APACHE_GET, SOCKET_GET, etc
SMS	SEND_TEXT_MESSAGE, etc
Premium service	**Category**
SMS	SEND_TEXT_MESSAGE, etc
CALL	OUTGOING_CALL, etc
Encryption	**Category**
ENCRYPT	ENCRYPT_AES, ENCRYPT_DES, etc

- broadcast means that the behavior is triggered when a broadcast message is received;
- listener means that is is triggered when the registered listener captures a change on the device states;
- observer refers to the observer that is registered on ContentProvider. The observer triggers the behavior when the content provider is changed.

Permission features refer to the permission required for the malware to conduct malicious behaviors [97]. Android provides a permission-based mechanism to avoid the abuse of system sensitive operations. Many malicious behaviors in malware require certain permissions to achieve attack goals. For example, the permission android.permission.READ_PHONE_STATE is required to obtain the IMEI code of the device via invoking the method getDeviceId. In general, information leakage requires the permission for accessing and sending out the information. Similarly, phishing also requires such permissions. Financial charge requires the permission for sending SMS to a number that binds to premium rate services. The recent extortion attack needs to access the public files on device with the permission android.permission.WRITE_EXTERNAL_STORAGE.

Behavior features refer to the malicious behaviors conducted by the attack, to which trigger and permission features are all assistant [57]. Behavior features are the core attack features that mostly link with the modularized malicious code. For the four types (privacy leakage, financial charge, phishing, and extortion) of attacks shown in Section 2.2, there are four types of behavior features, respectively. For each type of behavior features, several steps need to be carried out for the success of the attack. For each attack step, it may have several sub-features that represent different implementations. For instance, in privacy leakage, two steps are carried out: obtaining the privacy (i.e., feature source) and leaking the privacy (i.e., feature sink). For feature source, there are multiple candidate sub-features (SMS information, device information, etc.). Similarly, for feature sink, the leaked privacy can be sent out by SMS or HttpRequest.

Note that the partial feature model in Fig. 5.2 mainly illustrates the high-level organization of these features. Each feature at the bottom level in Fig. 5.2 may have several sub-features, e.g., feature Source has 11 variant sub-features in an *Or* relationship. Each variant feature in Table 5.1 may have several sub-features of different implementations (modularized code) in an *Alternative* relationship. Interested readers can refer to our tool website [163] for the complete feature model, the full list of CTCs among the features.

```
public void onCreate(Context context, Intent intent){
    if (intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED")){
        final Bundle bundle = intent.getExtras();
        if (bundle != null) {
            final Object[] pdusObj = (Object[]) bundle.get("pdus");
            for (int i = 0; i < pdusObj.length; i++) {
                SmsMessage currentMessage = SmsMessage.createFromPdu((byte[]) pdusObj[
    i]);
                sb.append(currentMessage.getDisplayMessageBody()).append("&");}
        }
    }
    SmsManager sm = SmsManager.getDefault();
    sm.sendTextMessage(number, null, message, null, null);
}
```

FIGURE 5.3: The modularized code of sending token via SMS ($D1$)

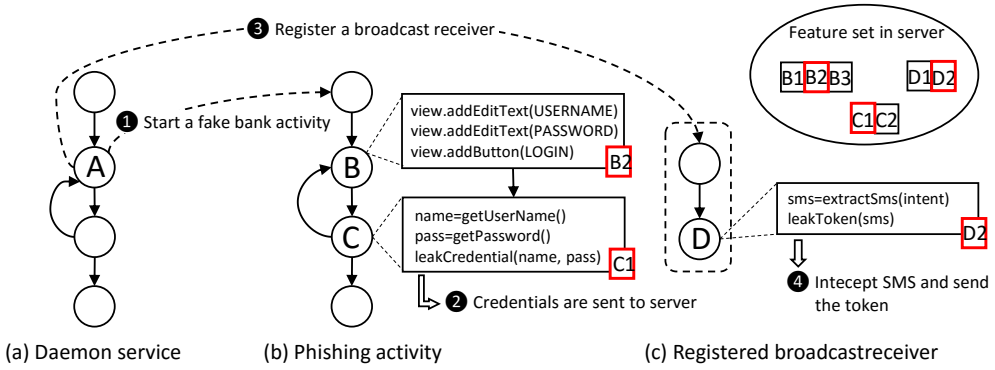

FIGURE 5.4: A running example of Mystique-S

5.3.2 Feature Modularization

The code of AFs is modularized into code units of various granularity, ranging from several packages to a single method. The phishing AF usually contains the largest number of lines of code (LOC), as it has the faked GUI or functionalities to deceive the users. Hence, the corresponding code of phishing attack can be close to the genuine app, with the LOC up to a reasonably large number. In contrast, the implementation of financial charge (or adware) can be just several lines of code and easily modularized into a method. For example, in Fig. 5.3, we show the modularized code for sending the token by SMS (*D1*). The token is intercepted by registering a BroadcastReceiver and listening the incoming SMS messages and then sent out via an SMS message to a specific number via SmsManager.

5.4 Running Example and System Overview

5.4.1 A Motivating Example

Fig. 5.4 depicts an exemplar of malware service that dynamically loads malicious code from a remote server[1]. The basic idea is to disguise the client app as a benign app that tricks users into entering credentials and then intercepts the SMS with two-factor token. The basic steps are executed as follows:

- After the client app is installed on user device, it starts a daemon service to communicate with the service provide of malware. It collects and sends the user information (e.g., hardware and software information of the device) to the server, and receives the malicious payloads from the server.
- After the malicious code is delivered to the client, the daemon service starts a fake bank activity from the component *A* inside (**Step 1** in Fig. 5.4). In the life-cycle of the phishing activity, two code snippets are instrumented into the component *B* and *C*, respectively.
- The code in *B* is to change the view of activity to mimic the specific bank app, and the code in *C* is to get the entered credentials and send them to the server (**Step 2**).
- Last, the daemon service registers a broadcast receiver to listen to incoming SMS messages (**Step 3**). The SMS message that contains the two-factor token (the key for two-factor authentication) is leaked to the attacker (**Step 4**).

As shown in Fig. 5.4, for the same attack step, there may exist various implementations, which are also regarded as candidate attack features. For example, for the phishing attack in Fig. 5.4, there exist three attack feature candidates (i.e., different views) of phishing attack (*B1, B2, B3*). For feature *LeakCredential* in component *C*, there are two attack feature candidates: sending credentials by Apache connection (*C1*) and sending

[1]The original version of the example malware is found in March 2016 [164], but it is neither service-oriented nor dynamically loaded.

them by SMS (*C2*). For the feature *LeakToken* in component *D*, there are two candidates: sending token via SMS (*D1*) and sending token via socket (*D2*). For simplicity, in this example, we just show two or three candidate attack features for each attack step, and also omit the finer-grained attack features of the source and sink operations at steps 2 and 3. Types and granularity of attack features are explained in Section 5.3.

For this example, three tree constraints (TCs) and five cross-tree constraints (CTCs) need to be satisfied. For example, TC_2 means if $LeakCredential$ is selected, at least one of $C1$ and $C2$ must be selected, and vice versa. CTC_3 means the selection of *LeakToken* requires the selection of permission feature $P3$.

$TC_1 : B1 \vee B2 \vee B3 \Leftrightarrow Phish$	— *or* type
$TC_2 : C1 \vee C2 \Leftrightarrow LeakCredential$	— *or* type
$TC_3 : D1 \vee D2 \Leftrightarrow LeakToken$	— *or* type
$CTC_1 : C1 \Rightarrow P1\ (android.permission.INTERNET)$	— *require* type
$CTC_2 : C2 \Rightarrow P2\ (android.permission.SEND\_SMS)$	— *require* type
$CTC_3 : LeakToken \Rightarrow P3\ (android.permission.RECEIVE\_SMS)$	— *require* type
$CTC_4 : D1 \Rightarrow P2\ (android.permission.SEND\_SMS)$	— *require* type
$CTC_5 : D2 \Rightarrow P1\ (android.permission.INTERNET)$	— *require* type

The suitable attack features need to be automatically selected for the sake of a better success ratio of attack, given different user scenarios (e.g., model of device, OS version and installed AMTs). For example, if the device installs NORTON, attack features {*B2*, *C1*, *D2*, *P1*, *P3*} should not be selected. The reason is that NORTON reports suspicious apps based on the permission $P2$. If no AMT is installed, attack features (*B2*, *C2*, *D1*, *P2*, *P3*) are selected as $P2$ can be selected for short latency due to the immediate action of sending SMS messages.

5.4.2 System Architecture

MYSTIQUE-S is a framework of automated malware generation, which takes as input the client-end contextual information and outputs the user-tailored malicious code.

Based on the Android malware feature model (§ 5.3), MYSTIQUE-S automatically selects attack features according to the user scenario via linear programming (LP in § 5.5.3). Then, the selected attack features guide the model-driven generation of malicious code (§ 5.6). Last, the *payloads* are delivered to the user device and loaded dynamically (§ 5.6). Here, *payloads* refer to the generated malicious code and the corresponding *instructions* (i.e., the command for the client app to load the code of attack features in sequence).

Fig. 5.5 depicts the architecture of our tool, which contains three parts as we discuss below:

- **Client app.** Its task is to (periodically) collect the contextual information on the user device, receive malicious code and instructions from the server, and launch the attack by using the dynamic code loading mechanism (§ 5.6.3). As shown in Fig. 5.5, three critical modules are included in the client app: 1) *daemon service* interacts with the service provider and starts an attack once receiving the malicious code and instructions. 2) *dynamic instrumentation* deploys the malicious code in different components (e.g., Intent) of the client app, interprets the received instructions and acts accordingly. 3) *execution of malicious behavior* executes the instructions (loading the malicious code of multiple attack features in sequence). Finally, the execution results are fed back to the daemon service.

- **Service provider.** The service provider listens to the requests from the client app on installed devices. After receiving the user device information, it selects attack features and generates the corresponding payloads. Four modules are involved in the process: 1) *request listener* receives attack requests and initializes the automatic generation of payloads. 2) *LP-based feature selection* selects an optimal combination of attack features from the Android malware feature model. 3) *instruction generation* takes input as the selected attack features and generates the instructions by considering the context in the client app. One instruction, in the format of BDL that specifies the workflow of malware (§ 5.6.1), contains the execution context and the operation to execute. 4) *code generation* generates the malicious code by assembling the code of attack features according to the BDL.

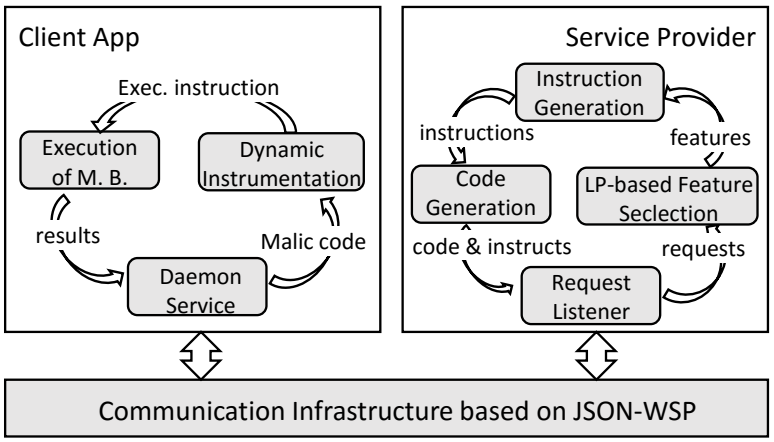

FIGURE 5.5: The overview of system

After the process, *request listener* sends the generated payloads to the daemon service on user device.

- **Communication infrastructure.** It provides a connectionless protocol that enables the asynchronous communication between the client app and the server. As an attack needs multi-round interactions between the client app and the server, the connection is not retained during the lifecycle of an attack for the sake of hiding the attack. Instead, the service provider will track the state where the attack proceeds. In addition, the exchange message follows the standard JSON-WSP [165] for a bidirectional communication (see § 5.6.3).

5.5 User-Tailored Attack Feature Selection

In this section, we explain how the user-tailored attack features are automatically selected by linear programming (LP). First, we show how to convert TCs and CTCs among features to inequalities for LP based constraint solving (§ 5.5.1). Then we define the malware generation goals (§ 5.5.2). Last, we resolve the attack feature selection problem via LP (§ 5.5.3), i.e., satisfying the inequalities and optimizing the objective functions.

5.5.1 Converting Features Constraints to Binary Inequalities

To select features and generate the products that satisfies the TCs and CTCs (defined in § 5.2.1) inside the feature model, Broek [166] adopted integer programming (IP) for the feature selection problem (i.e., initialization of valid product in [166]). Broek converted the TCs and CTCs into the integer inequalities, and then apply IP to resolve these inequalities. In this thesis, we further convert the TCs and CTCs into the inequalities of binary variables. Given a feature f, the binary value represented by the selection of f (denoted as $|f|$) is **1** if selected, and otherwise $|f|$ is **0**. According to the integer inequalities deduced in [166], we can further deduce the corresponding binary inequalities for the TCs and CTCs. In Table 5.2, we list the binary inequalities for different types of constraints.

5.5.2 Goals of Attack Feature Selection

Apart from the constraints to satisfy, we also need to define the design goals for malware generation. We propose three objectives to guide the attack feature selection: *aggressiveness*, *latency*, and *detectability*. As the results of attack feature selection, malware is getting more aggressive with shorter latency, but being less detectable. Given a solution $\vec{x}$, we represent it as a bit vector of all attack features, where $\{f_1...f_n\}$ denotes the set of n attack features. The objective functions are defined as follows.

1. *Aggressiveness*: to make the malware more aggressive, we want to minimize the number of attack features that are not selected. It is defined as: $\mathscr{F}_1(\vec{x}) = \sum_{i=1}^{n}(1 - |f_i|)$.

2. *Latency:* to shorten the time-delay in attack launching (e.g., leaking by SMS has less latency than leaking by Internet), we aim to minimize the total latency of all selected features. It is defined as: $\mathscr{F}_2(\vec{x}) = \sum_{i=1}^{n}(|f_i| \times l_i)$, where l_i denotes the latency of attack feature f_i.

TABLE 5.2: Binary inequalities for different types of constraints

Constraint Type	Binary Inequality
f and its *mandatory* sub-feature f'	$\|f'\| - \|f\| = 0$
f and its *optional* sub-feature f'	$\|f'\| - \|f\| \leqslant 0$
f and its *or* sub-features	$\forall\, i \in \{1, ..., n\}\ \ \|f'_i\| - \|f\| \leqslant 0$ $\sum_{i=1}^{n} \|f'_i\| - \|f\| \geqslant 0$
f and its *alternative* sub-features	$\forall\, i \in \{1, ..., n\}\ \ \|f'_i\| - \|f\| \leqslant 0$ $\sum_{i=1}^{n} \|f'_i\| - \|f\| \geqslant 0$ $\sum_{i=1}^{n} \|f'_i\| \leqslant 1$
f_1 *requires* feature f_2	$\|f_1\| - \|f_2\| \leqslant 0$
f_1 *excludes* feature f_2	$\|f_1\| + \|f_2\| \leqslant 1$
f_1 *iff* feature f_2	$\|f_1\| - \|f_2\| = 0$

3. *Detectability:* to increase the chance for malware to succeed, we minimize the probability to be detected by AMTs. It is defined as: $\mathscr{F}_3(\vec{x}) = \sum_{i=1}^{n}(|f_i| \times d_i)$, where d_i denotes the detection ratio of attack feature f_i if f_i is applied alone.

Intuitively, *Obj1* and *Obj2* are competing with *Obj3*, meanwhile *Obj1* and *Obj2* are mutually competing. For instance, having more attacks or shorter latency will lead to earlier and easier detection of the attack. Besides, having more attack features, which is desired, can lead to an undesired side effect of higher latency. With the feature constraints in Section 5.5.1 that are linear, the three objective functions are also linear. Hence, LP can be applied to resolve this optimization problem. Note that $l_i \in [0,3]$ and $d_i \in [0,10]$ are empirical values, according to our preliminary studies. For example, latency l is set to 1 for $C1$, and 2 for $C2$; detectability d is set to 3 for $C1$, and 4 for $C2$. More discussions on the setup of values of l_i and d_i can be found in Section 5.8.

5.5.3 Attack Feature Selection via LP

For the richness of possible solutions, we would not encode three objectives into one weighted objective for one time solving. Instead, we treat each objective equally and solve this *Multi-objective Optimization Problem (MOP)* using the Pareto dominance relation [138]. In MOPs, usually there exists no single solution that simultaneously optimizes all objectives. Hence, we are interested to find the *non-dominated solutions*. A solution is called non-dominated, if none of the objectives can be improved in value without degrading other objectives [138].

A k-objective optimization problem could be written in the following form (in our case, $k = 3$):

$$\textit{Minimize } \vec{\mathscr{F}} = (\mathscr{F}_1(\vec{x}), \mathscr{F}_2(\vec{x}), ..., \mathscr{F}_k(\vec{x})) \tag{5.1}$$

Subject to the inequalities on variables ($|f_1|...|f_n|$ in our case),
where $\vec{\mathscr{F}}$ is a k-dimensional objective vector, $\mathscr{F}_i(\vec{x})$ is the value of $\vec{\mathscr{F}}$ for i-th objective, and $\vec{x}$ is the feature set $\{f_1, ..., f_n\}$.

Technical innovation. To resolve MOPs, MOEAs are often applied [134, 167]. MOEAs are generally scalable, but it requires some evolution time. As heuristic search techniques, MOEAs cannot guarantee to find many non-dominated solutions. Traditionally, LP can only solve single-objective LP optimization. Considering the manageable feature size of the feature model (§ 5.3), we apply LP to resolve the MOPs in an analytic way.

The basic idea is that: we retain an objective as the goal function for optimization, and convert two other objective functions into constraints by setting the concrete bounds for them. To find more non-dominated solutions, we need to gradually adjust the bounds for these two objective functions.

Algorithm 4 depicts the main process of LP-based attack feature selection. At lines 1-3, the user information (e.g., the model of device, OS version and installed software) is

Algorithm 4: Linear programming guided feature selection

Input: $featureMdl$: the feature model of Android malware
Input: $userInfor$: the contextual information of user device
Output: $solutions$: a non-dominated solution set for feature selection
Output: $returnedSol$: a solution returned to guide malware generation

```
1  solutions ← ∅
2  upper_o2 = getInfor(userInfor), lower_o2 = 0
3  upper_o3 = getInfor(userInfor), lower_o3 = 0
4  for i = lower_o2; i ⩽ upper_o2; i = i+1 do
5      for j = lower_o3; j ⩽ upper_o3; j = j+1 do
6          const_o2 = convert(obj2, i), const_o3 = convert(obj3, j)
7          allConsts = TCs ∪ CTCs ∪ const_o2 ∪ const_o3
8          nondominatedSol = bintprog(allConsts, obj1)
9          solutions = solutions ∪ nondominatedSol
10 returnedSol = solutions.First()
11 for sol ∈ nondominatedSol do
12     if aggregatedObj(sol) < aggregatedObj(returnedSol) then
13         returnedSol = sol
14 return returnedSol;
```

analyzed via function $getInfor()$ and the searching bound for *obj2* and *obj3* are suggested. For the example in Section 5.4.1, if the user device installs many AMTs and the latest Android version, the malware should have a low detection ratio (a small ratio of the theoretic upper bound, e.g., $10\% \times \sum_{i=1}^{n} d_i$), and can tolerate a little high latency (a large ratio of the theoretic upper bound, e.g., $50\% \times \sum_{i=1}^{n} l_i$). At lines 4-9, we gradually adjust the upper bounds of *obj2* and *obj3* and get the corresponding solutions. At line 8, $bintprog(allConsts, obj1)$ is the LP solving function that optimizes $obj1$, subject to the constraints of inequalities in $allConsts$. Finally, it reaches the termination condition (i.e., the upper bounds) and gets the candidate solutions into $solutions$.

For the constraints of the example in Section 5.4.1, according to Table 5.2, we can convert these logical formula to the inequalities for LP solving function $bintprog$. For the termination case of our example, line 6-9 get the following inequalities and perform the LP solving. Note that $|B1|$ returns 1, if $B1$ is selected; $O2C$ is the constraint converted from *obj2*, where $\sum_{i=1}^{n} l_i$ refers to the sum of latency of each feature; $O3C$ is converted from *obj3*, where $\sum_{i=1}^{n} d_i$ refers to the sum of the chance of each feature to be detected.

Minimize $\vec{\mathscr{F}} = (\mathscr{F}_1(\vec{x}))$, where $\vec{x}$ is the set of all features
Subject to: $TC_1 \wedge TC_2 \wedge TC_3 \wedge CTC_1 \wedge ... \wedge CTC_5 \wedge O2C \wedge O3C$
$TC_1 : |B1| \leqslant Phish, |B2| \leqslant Phish, |B3| \leqslant Phish, |B1|+|B2|+|B3| \geqslant Phish$
$TC_2 : |C1| \leqslant LeakCredential, |C2| \leqslant LeakCredential, |C1|+|C2| \geqslant LeakCredential$
$TC_3 : |D1| \leqslant LeakToken, |D2| \leqslant LeakToken, |D1|+|D2| \geqslant LeakToken$
$CTC_1 : |C1| \leqslant |P1|$ $CTC_2 : |C2| \leqslant |P2|$
$CTC_3 : |LeakToken| \leqslant |P3|$ $CTC_4 : |D1| \leqslant |P2|$
$CTC_5 : |D2| \leqslant |P1|$
$O2C : 0 \leqslant \mathscr{F}_2(\vec{x}) \leqslant 50\% \times \sum_{i=1}^{n} l_i$
$O3C : 0 \leqslant \mathscr{F}_3(\vec{x}) \leqslant 10\% \times \sum_{i=1}^{n} d_i$

At lines 11-13, among the candidate solutions, we combine several objectives into an aggregated one, by normalizing the ranges of objectives and assigning them with different weights via function $aggregatedObj()$ at line 12. At lines 12-13, we iterate all candidate solutions and identify the optimal solution according to the weighting scheme. In addition, in practice, we refine the returned optimal solution by applying some extra constraints, which are not from the feature model, but from the observations on AMTs and attack features. For example, if NORTON is installed on the device, feature $P2$ android.permission.SEND_SMS should not be selected — NORTON reports the third-party app as suspicious if it requires $P2$. In other scenarios, if no AMT is installed on the device, attack features (*B2*, *C2*, *D1*, *P2*, *P3*) are selected as $P2$ can be selected for the short latency of sending SMS immediately.

We clarify that to utilize the user contextual information, the relaxed LP approach is proposed to run LP solving for multiple times. With more candidate solutions, the variety of selected attack features (and the generated code) is improved, preventing the signature- or clone-based detection. Instead, directly combining 3 objectives into an aggregated one and solving it once just yields one solution, which impairs the variety and the unpredictability of the selected attack features.

5.6 Dynamic Generation and Execution of Malicious Code

After the server conducts attack features selection via linear programming, we show how to assemble the corresponding code of attack features via a model-driven way (§ 5.6.1 and § 5.6.2). Then, we explain how the generated malicious code is sent to the client app via JSON-WSP. Last, it is dynamically loaded and executed at the client end (§ 5.6.3).

5.6.1 Behavior Description Language

Semantics of the selected attack features is represented in a modeling language, named Behavior Description Language (BDL). The BDL representation for the attack features is more implementation oriented. BDL is used for two purposes: on the server side, it bridges the gap between the malware feature model and the workable implementations; on the client side, it assures that behaviors of attack features are executed as designed.

Backus Naur Form of BDL. We present the partial BNF of BDL in Fig. 5.6 (refer to [133] for the complete definition of BDL). An attack can be divided into several subsequential operations, i.e., $\langle ATTACK \rangle ::= \langle FUNCTION \rangle (' \rightarrow ' \langle FUNCTION \rangle)^*$. Hereby, $\langle FUNCTION \rangle$ is the basic step (building block) for an attack, and it denotes the operation to execute as well as the execution context. One function consists of three elements — $\langle COMPONENT \rangle$, $\langle POINTCUT \rangle$ and $\langle OPERATION \rangle$, where $\langle COMPONENT \rangle$ denotes the component, the building blocks of Android apps, $\langle POINTCUT \rangle$ denotes the methods where malicious behaviors are located, and $\langle OPERATION \rangle$ denotes the operation of malicious behaviors. The component and method together identify the execution context for this operation.

Connection between feature and BDL. As the direct assembly of code of the selected attack features may not yield a workable (no compilation or runtime error) malicious code. Hence, BDL is required to bridge the gap between the selected attack features and the code implementation by adding the execution context of attack features and auxiliary behavioral operations in implementation.

$$
\begin{aligned}
&\langle ATTACK\rangle && ::= \langle FUNCTION\rangle('\rightarrow' \langle FUNCTION\rangle)^* \\
&\langle FUNCTION\rangle && ::= \langle COMPONENT\rangle' ::' \langle POINTCUT\rangle' ::' \langle OPERATION\rangle \\
&\langle COMPONENT\rangle && ::= ' ACTIVITY' |' SERVICE' |' BROADCAST\_RECEIVER'... \\
&\langle POINTCUT\rangle && ::= ' POINTCUT\_ONCREATE' |' POINTCUT\_ONSTART'... \\
&\langle OPERATION\rangle && ::= \langle SOURCE\_SIG\rangle \mid \langle ENCRYPT\_SIG\rangle \mid \langle PHISH\_SIG\rangle...
\end{aligned}
$$

Figure 5.6: Parts of BNF for BDL

Conceptually, among the selected attack features, each behavior feature relates to one $\langle FUNCTION\rangle$ in BDL. As behavior feature is defined at the atomic behavior level (one step of the attack), its corresponding code is usually modularized into the code unit of method. The modularized code of feature conceptually links to one $\langle OPERATION\rangle$. Hence, assembling modularized code of features essentially requires to describe an $\langle OPERATION\rangle$ with the proper $\langle COMPONENT\rangle$ and $\langle POINTCUT\rangle$. For example, one attack of privacy leakage is to steal users' SMS messages. According to the feature model, it needs a $\langle FUNCTION\rangle$ to get SMS messages (i.e., source), and a $\langle FUNCTION\rangle$ to send them out (i.e., sink). These two steps comprise this attack. The code method of source is an $\langle OPERATION\rangle$, and this method is invoked in some $\langle COMPONENT\rangle$. The source operation also needs a permission feature android.permission.READ_SMS, and the behavior need to be started in some $\langle POINTCUT\rangle$ — e.g., from bootup of an app (i.e., trigger feature main) or from a change event of a Content Provider (i.e., trigger feature observer).

Hence, BDL can provide details on: the component of activity or service, the method where the malicious code is injected and executed; the data flow from source to sink, using Android lifecycle and Inter-Component Communication (ICC).

5.6.2 Model Driven Malicious Code Generation

In MYSTIQUE-S, we have set some rules for automated generation of BDL for selected attack features, including various commonly-used source-sink patterns [21], and information flows for phishing attack.

The service provider further interprets BDL to generate the corresponding malicious code. As the malicious code is dynamically loaded and executed in the client app, MYSTIQUE-S will not bind or invoke the code snippets of attack features at server side. Hence, the generated malicious code includes two parts: the declaration of code for AFs (in the format of Java method), and the invocation method to attack features.

An illustrative example. For the example in Fig. 5.4, it is a composite attack with privacy leakage and phishing. As the phishing feature can only be deployed in the main thread of an activity, it is assigned to the context of ACTIVITY::ONCREATE. The acquisition of incoming SMS messages need to be done in the context of a registered broadcast receiver. Thus, the selected attack features (i.e., *B2*, *C1*, *D2*) in § 5.4.1 have the corresponding BDL:

$$
\begin{aligned}
&ACTIVITY::ONCREATE::PHISH() \\
&\rightarrow ACTIVITY::ONCREATE::SINK(HTTP :: APACHE\_POST, CREDENTIALS) \\
&\rightarrow BROADCAST\_RECEIVER::ONRECEIVE::SOURCE(SMS::INCOMING\_SMS) \\
&\rightarrow BROADCAST\_RECEIVE::ONRECEIVE::SINK(HTTP::SOCKET\_POST, \\
&\quad LOCAL\_VARIABLE)
\end{aligned}
$$

Based on the above BDL, MYSTIQUE-S generates the malicious code in Fig. 5.7. Lines 3-14 provide the declarations for these features, and line 15-22 present the invocation to these declarations. In method "operateOn", it defines the statements (i.e., the invocations to specific feature declarations) as the instruction of attack for different steps.

5.6.3 Dynamic Loading and Execution of Malicious Code

Malicious code is dynamically loaded and executed in the client app. The process relies on two mechanisms as below.

Single-step loading via JSON-WSP. JavaScript Object Notation Web-Service Protocol (JSON-WSP) [165] is a web-service protocol that uses JSON for service description. We use JSON-WSP to exchange messages between client app and the server.

```
class Task{
  /* Feature declarations  */
  // code of phishing feature B2
  void phishing(){ ... }
  // "Sink" of feature C1, send credentials by Apache conn.
  String sendCredential(String data){... }
  // "Source" code of feature D2, read incoming SMS.
  String getIncomingSms(){ ... }
  // "Sink" code of feature D2, send token by Socket conn.
  String sendToken(String data){...}

  /* The invocation to features */
  Object operateOn(String comp, String met){
    if (comp=="ACTIVITY"&&met=="ONCREATE") {
       phishing();
    }else if (comp=="BROADCAST_RECEIVER"&&met=="ONRECEIVE"){
       sendCredential(getIncomingSms())  ;
    }
   ...}
}
```

FIGURE 5.7: Generated code for the selected attack features (*B2*, *C1* and *D2*)

Initially, the service provider generates a sequence of instructions to execute an attack. The client app queries and receives from the server an instruction each time, named *single-step loading*. The main part of instructions contains the type of instructions and the content of the instructions, in the format of {"command":"", "value":""}. There are two types of instructions — *download* that indicates the address of the payload to download, and *execute* that provides a serial of operations in BDL. For the running example, the first instruction received by single-step loading is a *download* instruction to download the malicious code, the following *execute* instruction is to execute the behaviors defined in the BDL. (§ 5.6.2).

Dynamic execution via reflection. MYSTIQUE-S employs Java Reflection to dynamically execute the malicious code. Similar with the idea of XPOSED [168], MYSTIQUE-S injects a small code snippet (shown in Fig. 5.8) into each execution context of Android app. The code then checks the payloads whether there is a task to execute in this current context. As the payloads (e.g., operateOn in Fig. 5.7) define the operations to do in different contexts, the malicious behaviors are dynamically loaded into a specific context. In Android, reflection is based on the class DEXCLASSLOADER which can load *dex* files and read the included class files. As shown in Fig. 5.8, the client app needs to create an instance of DexClassLoader by specifying the location of the *dex* file. The class loader is used to instantiate the target class and thereby the target method.

```
1 DexClassLoader loader = new DexClassLoader("[DEX_FILE]", "[CACHE_FILE]", "[LIB_PATH]",
      "[CLASS_LOADER]");
2 Class clz = loader.loadClass("Task");
3 Object obj = clz.newInstance();
4 Method mtd = clz.getDeclaredMethod("operateOn", "[COMP]", "[POINTCUT]");
5 mtd.invoke();
```

FIGURE 5.8: A simple example of using reflection mechanism

5.7 Evaluation

MYSTIQUE-S is implemented in about 4,187 lines of Java code (23.9% for the client app, 76.1% for the service provider, and modularized attack feature code is not included). It adopts CPLEX [169] for solving LP. Considering the dynamic attack, experiments are conducted on dynamic Analysis Tools (DATs) or real devices installed with AMTs; the service provider is deployed on a workstation running on Ubuntu 14.04 with Intel Xeon(R) CPU E5-2697 and 64G memory. We aim to answer the following research questions.

1. Are the modularized attack features valid? Is the dynamically assembly malicious code workable at runtime?
2. Can the mainstream AMTs and online vetting process detect the malware dynamically generated by our tool?
3. Is MYSTIQUE-S adaptive to the different attacks in real cases?

Evaluation subjects. To evaluate the evasiveness of the dynamic attack and audit the AMTs, we select several state-of-the-art AMTs for detection in Table 5.3 and 5.4.

5.7.1 RQ1: Validity of Generated Malicious Code

In this section, we evaluate the validity of MYSTIQUE-S. Specifically, we conduct experiments to show the validity of malicious code that is generated from each AF. Further, we evaluate the service-oriented communication mechanism between the server and the client app.

Among the 92 attack features introduced in § 5.3.1, we identify 44 behavior features. For each behavior feature, we select its required permission features and trigger features, and generate the BDL representation. MYSTIQUE-S generates the corresponding malicious code according to the BDL. Then we repackage the malicious code into a blank Android app to wrap it as malware. Finally, we execute the malware on the emulator to verify whether the carried malicious code can be successfully executed. The results show that malicious code can fulfill its malicious intent, e.g., leaking information, extortion. In this experiment, we confirm that each behavior feature, as single building block, is valid and workable on its own.

To confirm the validity of the generated malicious code, a *honeypot* is set up to receive the report of a successful attack (e.g., the stolen information is sent to the honeypot) in the experiment. Our honeypot has successfully received the response from emulators or experimental devices. It proves that our generated malicious code works in practice, which encourages us to conduct user studies on real devices (§ 5.7.2).

During the communication between the client app and the service provider, multiple sequential instructions are exchanged to complete an attack. The bidirectional communication is asynchronous, which means that every time the client app may receive and execute only one individual instruction. To guarantee the client app has obtained all necessary malicious code and instructions, MYSTIQUE-S employs *periodical querying* in the client app and *state retaining* in the service provider. The daemon service in client app will periodically enquire service provider to check: 1) it is alive; 2) what to do in the next step. This mechanism avoids the tense work (e.g., high network traffic and high memory usage rate) with launching an attack, and thereby reduce the probability of being perceived by users. After identifying the attack to launch with LP, the service provider retains the state where the attack proceeds. In our experiments, we set the time interval as 30 minutes for periodical querying. Results show that this mechanism can tolerate the loss of Internet connection, and restore the attack state after the client app is reconnected to Internet.

TABLE 5.3: The detection results of ODTs, where ✓ means "passed" and ✗ means "detected"

Tool	DS#A	DS#B	Tool	DS#A	DS#B
FLOWDROID	✓	✗	ICCTA	✓	✗
DROIDSAFE	✓	✗	NORTON	✓	✓
AVG	✓	✓	AVAST	✓	✓
BITDEFENDER	✓	✓	ESET	✓	✓
KASPERSKY	✓	✓			

5.7.2 RQ2: Auditing the AMTs on Real Devices

We have evaluated the resistance of generated malicious code to the detection in three aspects: offline detection tools, dynamic analysis tools and AMTs installed on Android devices.

5.7.2.1 Resistance to Offline Detection Tools (ODTs)

To evaluate the evasiveness of the client app against ODTs, we choose several state-of-the-art static analysis tools and AMTs from VIRUSTOTAL. To evaluate the efficacy of this dynamic and optimal selection of attack features, we conduct an experiment that uses the client app with/without the payloads, respectively. As shown in Table 5.3, column *DS#A* shows the results of scanning the client app without payloads; column *DS#B* shows results of scanning the client app with payloads. Here, payloads are the malicious code generated according to the 44 behavior features.

Based on observation from Table 5.3, it is concluded that MYSTIQUE-S can effectively bypass the detection of ODTs. Generally, static analysis collects the evidences in the *apk* file for detection. However, MYSTIQUE-S only dynamically loads malicious code in an attack, and it does not store any malicious code in the *apk* file. Hence, it has a very low probability of being detected by offline ODTs and AMTs.

5.7.2.2 Resistance to Dynamic Analysis Tools (DATs)

We deploy three state-of-the-art DATs to evaluate the evasiveness of MYSTIQUE-S. These three tools are listed below:

- **DROIDBOX**[2] automatically intercepts and modifies API calls made by a targeted app. It captures the behaviors of apps at runtime, e.g., information leakage, cryptographic operations, the invocations of Android APIs and etc.
- **DROZER**[3] allows you to search for security vulnerabilities in apps and devices by assuming the role of an app and interacting with the Dalvik VM.
- **TAINTDROID** [25] can track how apps use sensitive information via *taint analysis*. It has hooked several transfer channel including memory, file system, and event dispatch.

We construct 22 attacks (requesting specific permissions) of privacy leakage with regard to the types of sensitive information, 1 attack of premium service, 3 attacks of phishing, and 1 attack of extortion. DROIDBOX can successfully capture many behavior logs of the client app, for example, the download of malicious payload, the acquisition of contact and SMS, the operation to send SMS messages (perhaps to a premium rate number) and the cryptographic operation. However, it still needs manual efforts to confirm whether these behaviors are malicious or not. In comparison, DROZER can only identify the started Android components and the acquired permissions of the client app. Since TAINTDROID only targets privacy leakage of apps, it only detects 10 attacks (45.5%) of privacy leakage in our experiment, while it fails to detect other kinds of attacks.

Summary. The DATs can effectively detect attacks via dynamically loaded malicious code, compared to static analysis. It is reasonable because dynamic analysis can capture the runtime information, which can facilitate the understanding of current app operations. However, it has two issues that impede its practical use: *low scalability* that

[2]https://github.com/pjlantz/droidbox
[3]https://labs.mwrinfosecurity.com/tools/drozer/

makes it costly to detect a huge amount of apps, especially for the Android app stores; *high dependency* that makes it impossible to deploy it on real devices, as DATs usually rely on an in-depth instrumentation or modifications to Android OS.

5.7.2.3 The DR of Anti-virus

Due to the aggressiveness of the malware, we cannot conduct a large scale user study. We manage to have 16 volunteers install the client app on their devices. Before the experiments, they need to have at least one AMT installed on their device. We also assure them that the possible attack is just proof of concept (POC), e.g., leaking IMEI, leaking number of contacts, leaking a file's name and size only, deleting the copied one of a user file, etc. We replace the code of aggressive attack features (e.g., encryption) with that for POC. The profiles of devices and the detection results are presented in Table 5.4. Attack vectors for each device are selected by *LP-based attack feature selection* module. Details can be found at [133].

Evasiveness of malware. Generally, MYSTIQUE-S can easily bypass the scanning of most of AMTs shown in Table 5.4. Column *Inst.* means the scanning results of AMTs just after installation; column *Runt.* means whether AMTs give alerts when the attack is in progress; column *Succ.* means whether attacks succeed on the device.

As the attack is conducted by dynamically loading malicious code from the remote server and executing it locally, most AMTs fail to identify the maliciousness of client app after installation. There are only three AMTs that report the installed app as suspicious — 360 SECURITY, AVAST and NORTON.

Interestingly, in Table 5.4, the client app passes the scanning of 360 SECURITY on Nexus 6P, while it is detected by 360 SECURITY on Nexus 5. The detection capability in latest Android OS is even degraded in some cases. We speculate that some AMTs such as 360 SECURITY requests *root* permission to perform an in-depth scanning. So they even exploit n-day or zero-day vulnerabilities for rooting the user device. However, the latest Android OS (i.e., 6.0) fixes all known vulnerabilities and increases the difficulty in rooting. In reality, this weakens the detection capabilities of these AMTs. In addition,

TABLE 5.4: The detection results of AMTs on real devices, where column ✓ means "passed" and ✗ means "detected"

Phone Model	OS	SDK	AMTs	Inst.	Runt.	Succ.
Nexus S	3.0.1	11	McAfee	✓	✓	Y
Nexus 4	4.0.1	24	Bitdefender	✓	✓	Y
Nexus 5	5.0.1	21	360 Security	✗	✓	Y
Nexus 6P	6.0.1	23	360 Security	✓	✓	Y
Nexus 6P	6.0.1	23	Norton	✗	✓	Y
Samsung Note 3	5.0	21	Kaspersky	✓	✓	Y
Samsung Note 4	5.1.1	21	AVG	✓	✓	Y
Samsung Galaxy 4	4.4.2	19	Lookout	✓	✓	Y
Samsung Galaxy 5	4.4.2	19	CleanMaster	✓	✓	Y
Samsung Galaxy 6	5.0.2	21	AVG	✓	✓	Y
Huawei P8	5.0.1	21	AntiVirus	✓	✓	Y
Huawei Honor 7	5.0.2	21	Avast	✗	✓	Y
Nexus 6P	6.0.1	23	Avast	✓	✓	Y
Asus Zendfon Selfie	5.0.2	21	None	✓	✓	Y
Xiaomi MI 2	5.0.2	21	Avira	✓	✓	Y
Xiaomi Note 2	5.0	21	Baidu	✓	✓	N

NORTON reports our client app as suspicious. In further testing, we find that NORTON also reports many commonly used apps (which are normally regarded as benign) as suspicious, e.g., Facebook, GrabTaxi and Line. The reason is that NORTON employs a strict detection mechanism that gives many false positives. Note for the three alerted cases by AMTs, the attacks still succeed.

No matter whether AMTs give alerts after installation or at runtime, we confirm the attack results by checking whether the honeypot (§ 5.7.1) receives the attack response. We find the attack succeed on 15 out of 16 devices, while fails on Xiaomi Note 2. Further inspection shows this Xiaomi phone has compatibility problem with the client app that causes the failure of attacks.

Transparency of malware. We collect the feedback of user experiences from the 16 volunteers. They cannot notice the malicious behaviors of the client app, without any obvious symptom (e.g., high network traffic and high CPU consumption) observed. Hence, MYSTIQUE-S can silently conduct the malicious behaviors specified by the remote server while causing no attention of users. We attribute this to the adoption of LP-based attack feature selection for different user scenarios, which optimizes between the number of selected attack features, the chance to be detected, and the latency (overheads) of the attack.

5.7.3 RQ3: Generating Recent Attacks in Real Cases

To show the adaptivity of our tool, we combine the attack features to constitute the recent popular real-world attacks on Android.

5.7.3.1 Hacking Online Banking

Recently, numerous customers of Australia's largest banks are the victims of a sophisticated Android attack that steals banking details and thwarts two-factor authentication security. Our running example originates from this attack. Customers of mobile banking apps are at risk from the malware, which hides on infected devices waiting until users open legitimate banking apps. The malware then superimposes a fake login GUI over the top for intercepting usernames and passwords. The malware can mimic up to 20 mobile banking apps from Australia, New Zealand and Turkey, as well as login GUIs for PayPal, eBay, WhatsApp and etc.

Attack prerequisites. The following conditions need to be satisfied before attacking: **p1**, the specified malware is installed and started on victims' devices; **p2**, the malware is granted with sufficient permissions, including (android.permission.INTERNET) and (android.permission.RECEIVE_SMS); **p3**, the banking app employs the mechanism of two-factor authentication which needs send verification code to the register phone. The client app of MYSTIQUE-S can ride on some benign apps using "repackaging" [63]. In § 5.7.2, we show that the client app can easily evade the detection by AMTs, which

guarantees the **p1**. To satisfy **p2**, the client app asks for the necessary permissions (defined in Manifest file), which can be granted at installation (before Android 6.0) or at runtime (since Android 6.0). To satisfy **p3**, we mimic the login GUIs of the banking apps, such as CitiBank.

Attack vector. According to user's installed mobile banking apps (e.g., CitiBank), the user-tailored attack features (including the phishing feature for CitiBank login GUI) are selected. The service provider then generates the malicious payloads, consisting of malicious code and commands to execute. The malicious code can be referred to Fig. 5.7, and the commands in BDL can be referred to § 5.6.2.

Damage of attack. We have distributed this attack to 5 Android phones, from Android 4.0 to Android 6.0, and successfully collect the credentials and two-factor authentication. We discuss the possible damage from two aspects: the value of attack target and the user awareness of the attack. Once the bank account has been hacked, the attacker can obtain direct benefits from the victim which can cause a huge damage to the victims. From the perspective of users, there is no perceivable difference between the benign and phishing app, as Android activity as well as the views on it provide almost no hints for manual authentication. Unlike the phishing website that uses the fake URLs, careful users can spot some hints to authenticate. Therefore, it easily escapes from the awareness of victims.

5.7.3.2 Extortion app — Simplocker

Since the extortion malware *Simplocker* was found in 2014, ransomware has been swarming into the mobile app stores [170]. After launch, $Simplocker$ starts to encrypt files in a background thread. The encrypted files can be any format, and the encryption is by AES cipher. However, the encryption key is hard-coded in the binary file, which can be used to decrypt the files. It is believed that *Simplocker* is just a proof-of-concept or an early development version of more serious and complicated variants of ransomware.

Attack prerequisites. The attack needs to meet such prerequisites: **p1**, the malware is installed and started on user devices; **p2**, the malware is granted with sufficient permissions, e.g., the permission (android.permission.WRITE_EXTERNAL_STORAGE) to access to the storage. Same to the first case, MYSTIQUE-S satisfies **p1** and **p2**.

Attack vector. After installed, MYSTIQUE-S collects the information of the user device. If many important files are found on the device (e.g., many new taken photos or created user files), the user-tailored attack features (e.g., encryption, deletion) are selected. As the BDL below, four attack features are selected for this attack, and there are three constraints for these four features. Normally, the permission android.permission.INTERNET is acquired by default, which ensures the downloading of malicious payload. The features are deployed into the main thread of the daemon service, which can be represented as INTENT_SERVICE::MAIN.

Features:

encryption, deletion, android.permission.WRITE_EXTERNAL_STORAGE

android.permission.INTERNET (for downloading payload)

Constraints:

$encryption \wedge deletion \Leftrightarrow extortion$

$encryption \Rightarrow android.permission.WRITE\_EXTERNAL\_STORAGE$

$deletion \Rightarrow android.permission.WRITE\_EXTERNAL\_STORAGE$

BDL:

$INTENT\_SERVICE :: MAIN :: ENCRYPT(CIPHER, FOLDER)$

$\rightarrow INTENT\_SERVICE :: MAIN :: DELETE(FOLDER)$

Execution of the payloads generated from the BDL above performs the *encryption* on a certain folder, and delete it.

Damage of attack. This attack is distributed via MYSTIQUE-S, which is started by the client app. In this experiment, we use the AES to encrypt the specify folder and then delete the original files. The extortion attack can severely damage users' information properties. The target files, which are encrypted with a unknown cipher, may be very important to the victims. In addition, the extortion attack can optionally have the AF

sink, if the user device has 4G connection. This operation can further cause the leak of users' privacy.

5.7.3.3 Miscellaneousness

Spamming: $INTENT\_SERVICE :: MAIN :: SINK(SMS, LOCAL\_VARIABLE)$
$(\rightarrow INTENT\_SERVICE :: MAIN :: SINK(SMS, LOCAL\_VARIABLE))*$
Privacy: $INTENT\_SERVICE :: MAIN :: SOURCE(CONTACT :: CONTACT)$
$\rightarrow INTENT\_SERVICE :: MAIN :: SINK(HTTP, LOCAL\_VARIABLE)$
Privilege escalation: $INTENT\_SERVICE :: MAIN :: RUN(SHELL)$

MYSTIQUE-S can easily configure and generate a variety of attacks. For example, spamming is the kind of attacks which is annoying and exhaustive in recent years [171]. This attack can be easily achieved by conducting frequently *sink* operation. Hence, the SMS spamming can be represented with the BDL as above. MYSTIQUE-S can easily deploy the attack of privacy leakage using various source-sink patterns, which involve 11 types of sensitive information such as contact and SMS (Full details of sensitive information can be referred to [133]). As the above BDL, the client app can obtain the contact on the current device. In addition, MYSTIQUE-S can be further used to launch the attack of privilege escalation which needs shell code to root the device.

5.8 Discussion

Threats to validity. The internal threats to validity of evaluation stem from three aspects. First, regarding the completeness of attacks considered in this study, we just focus on the four types of attacks (§ 2.2) at this stage. In future, we will consider attacks such as privilege escalation that roots the device via vulnerability exploitation. Supporting privilege escalation will make MYSTIQUE-S similar to METASPLOIT on Android. Second, for the three goals of malware generation (§ 5.5.2), aggressiveness and detectability are security related, but latency is more on quality of service (QoS). In

future, we will consider other security or QoS related goals, e.g., to minimize the communication times and data size to exchange between the server and the client app. Last, for the values of d_i and l_i of a feature (§ 5.5.2), we now manually define these values according to our understanding of these attacks and results reported by the study [1]. An empirical study is required for the better setup of d_i and l_i for different attacks. The external threats are mainly two-fold. First, the malware samples for FODA are mostly from GENOME and DREBIN. Both of them contain many out-of-date malware, due to the everlasting malware evolution and creation. To ensure the timeliness of the feature model of malware, we have considered some recent samples of attacks of information leakage and extortion (§ 5.7.3). Another threat is about the availability of real devices and AMTs. More real devices need to be tested with more various AMTs.

To be or not be obfuscated? In this study, we do not further adopt the possible obfuscation techniques for the client app or the generated malicious code. Owing to the low detectability that we observed in the experiments (§ 5.7), it is not necessary to use extra obfuscation techniques for evading AMT detection. We also observe that existing AMTs do not sufficiently check the data that is received by a client app from the remote server at runtime. The rationale is that performing such check would impose a heavy burden on the performance. Besides, applying no obfuscation techniques eases the manual check of the generated malicious code for the experts. In reality, bytecode obfuscation techniques [126, 129] or wrapping payloads into native dynamic-link library (DLL) are applied for malware.

Possible enhancements for existing AMTs. To detect malware generated by MYSTIQUE-S, we propose two possible methods: 1) detection of C&C communications between the client app and the service provider; 2) malware detection based on bytecode clone detection. 3) a complete analysis of apps with dynamic payload. Actually, the first method is usually used for botnet or intrusion detection, but not a standard feature of AMTs. We find that AMTs normally cannot afford to check the data exchange of each app on Android. Firewalls often adopt the network traffic or DNS analysis [172, 173] to detect the C&C communication. Considering our tool as a testing framework rather than a real attack tool, we do not encode the C&C communication or use proxy strategies to prevent

the tracing of the service provider. So detection and hiding C&C communication is a topic different from this paper.

For the second method, the bytecode clones due to feature declaration (in Fig. 5.7) may exist in different payloads if some common attack features are selected in different attacks. The rationale is that reuse of the declared method of malicious code leads to the occurrence of code clones [16]. For example, feature $D1$ in Fig. 5.3 is commonly included as the sink in many attacks, which increases the chance for it to be detected. If no obfuscation is applied on $D1$, it can be easily exposed. Hence, if obfuscation is applied on the generated payloads, the clone-based detection method probably fails. To address this, an obfuscation-resilience clone detection technique is desirable for the detection of our generated malicious code with obfuscation.

Since the malicious code is enclosed in the payload which is dynamically loaded at runtime, the detection tools that only analyze host apps will definitely fail in detecting malware. Therefore, it is necessary to load the malicious payload and analyze it with the host app to reveal its maliciousness. There are several works [158, 174] which have employed both dynamic analysis and static analysis to detect malware.

5.9 Related Work

AMT auditing. AndroTotal [152] is a integrated framework to automatically test the detection capabilities of anti-virus tools. Christodorescu and Jha [149] leverage four types of obfuscation techniques to test the capabilities of commercial anti-virus tools. ADAM [153] employs several transformation techniques to generate polymorphic malware, and test 10 prestigious anti-virus tools. DROIDCHAMELEON [126, 129] collects three types of transformation attacks in Android, and the authors have used these attacks to audit the AMTs. Huang *et al.* [154] assess the detection capabilities of 30 top anti-virus tools from two aspects: malware scanning and engine updating. The study [175] also reports that existing AMTs are suspectable to dynamically loaded malware, using the existing malware.

Automated malware creation. According to the report by Zhou *et al.* [5], 86% of the 1260 samples are repackaged versions of benign apps with malicious payloads. Hence, repacking benign apps with malicious payloads is a cheap and fast way of malware creation. Based on the extra modules introduced by malicious payloads, the approach of module decoupling can effectively detect repackaged malware [63]. Recently, genetic programming has been applied to create malware in an automated way and evade the detection [147, 148]. Cani *et al.* [148] employ μ *GP* to automatically create new malware that is undetectable for anti-virus tools, and inject malicious code into a benign app to construct a Trojan horse. Aydogan and Sen [147] also adopt genetic programming to create Android malware. Different from the mutation operations on instructions of executables [148], Aydogan *et al.* mutate the CFGs (control flow graphs) that are extracted from smali code of GENOME malware [5]. Their experiments show that the new generated malware can easily bypass the detection of AMTs.

Evasive malware generation. Our work is also related to the generation of evasive or dynamically loaded Android malware. To evade the detection of AMTs [126, 129], DROIDCHAMELEON integrates three types of transformation techniques and generates obfuscated Android malware. Some evasion techniques used in DROIDCHAMELEON [126, 129] are identified as evasion features by Meng *et al.* in [1]. Hence, for the malware that contains malicious payloads at compile time before execution, the obfuscation [176] or evasion techniques (e.g., [126, 129]) are very helpful in failing the detection of anti-malware tools. To effectively evade the off-line detection (static detection before execution), Maier *et al.* [151] construct the *Divide-and-Conquer* attack, which combines fingerprinting and dynamic code loading to transform the existing malware. In addition to escape from off-line detection, Petsas *et al.* [150] propose three heuristics (static heuristics, dynamic heuristics and hypervisor heuristics) to fail dynamic analysis of Android malware. According to results of checking heuristics rules, the attack decides whether to launch the malicious payloads at run-time. In contrast, our malicious payloads are delivered from the remote server at runtime and can be purged after execution.

5.10 Conclusion

In this thesis, we propose to adopt the SPLE in order to modularize the common attack behaviors and construct the corresponding conceptual model (i.e., the feature model) for android malware. To provide a benchmark for dynamically loaded malicious code, MYSTIQUE-S adopts the DSPL techniques and makes the attack as a service, which facilitates the integration with other tools for AMT audit and penetration testing. We also evaluate the effectiveness of MYSTIQUE-S and the evasiveness of the generated malicious code on 16 real devices with 4 different recent attacks. In future, we will investigate the effectiveness of other attack and evasion features, such as obfuscating the generated malicious code. In addition, MYSTIQUE-S enables many studies on the malware generation and AMT auditing. Lastly, we will investigate the detection strategies for the malware generated by our tool on the fly.

6

A Large Scale Android Malware Analysis

6.1 Introduction

As the dominating mobile OS, Android is the main attacking target with a stunning increment of Android malware in last decade. According to the statistics in [3], there are around 3.3 million malicious apps from a variety of app markets to date. Due to the difficulty and overhead in security check of apps, third party Android markets are prone to be a hotbed of publishing and disseminating malicious apps. By leveraging these malicious apps, attackers can steal sensitive information, remotely control the device, subscribe premium rate services, blackmail the victims and so on [5].

To understand the nature of attacks, first of all, we need to know what is Android malware and what constitutes malware. Apart from the virus and trojanware, one type of

common-seen malware is the *piggybacked* malware that has the malicious payloads riding on a benign app [63]. According to the report [63], piggybacked malware accounts for 0.97% to 2.7% (the official GOOGLEPLAY Store has 1%). Besides the aforementioned types, there are numerous apps of *grayware*. Grayware is, actually an unwanted app or an adware, not as destructive as malware. For example, one grayware may pop up ads to make profits, or collect users' browsing histories to learn the behaviors. Hence, greyware also belongs to *potentially harmful apps* (PHAs), with malicious intention.

In the four million apps investigated by Symantec, about 33% of them can be categorized as grayware [177], while only 300,000 are strictly categorized as PHA. According to recent Google security report [178], only 0.15% of the apps installed from GOOGLEPLAY are PHAs. These reports fail to investigate the origin, increase, spread and of PHAs (see questions in Section 6.2.2). Aided by program analysis and statistic analysis techniques, we can answer a series of questions on how PHAs differ from benign apps and what makes PHAs. To sum up, all these questions help to understand this ultimate question: how risky and closely are the normal users exposed to the Android PHAs?

GENOME and DREBIN are two famous collections of Android PHAs. GENOME contains 49 PHA families and 1,260 PHA samples, but GENOME is only updated to PHA that emerged in 2012 [5]. DREBIN provides the 123,453 apps that are crawled from different app stores. Among the 123,453 apps, 5,560 samples are reported as PHAs according to their approach [20]. Due to the timeliness and transitory nature of PHA, it is not easy to conduct a large scale PHA analysis. Nevertheless, to comprehensively understand Android PHAs and answer the aforementioned questions (e.g., how many percents of apps are harmful, when the benign apps can become malicious), it is necessary to collect a large dataset of Android apps, including both benign apps and PHAs.

Insofar, we have collected 4,267,178 Android apps ranging from Sep. 2013 to Jun. 2016. These apps are collected from 32 different sources, including well-known PHA collections, ANDROZOO [179], GENOME [5], DREBIN [20] and VIRUSSHARE [47], and 28 different Android markets including GOOGLEPLAY. Among these 32 sources, 53.1% (17/32) of them are hosted in China, and 21.8% are hosted in US. According to the detection results of the mainstream anti-virus tools (AVTs), around 23.5% of

TABLE 6.1: The comparison to other large-scale Android security analysis

Tools	Scale(% Malware)	App		Malware			Market		Author		Vul.
		CQ1.1	CQ1.2	CQ2.1	CQ2.2	CQ2.3	CQ3.1	CQ3.2	OQ1.1	OQ1.2	OQ2
CHECK POINT [180]*	N.A.		✓	✓	✓		✓				
GOOGLE [178]*	35m (N.A.)		✓	✓	✓	✓					✓
MCAFEE [14]*	12m (100%)		✓	✓	✓		✓				✓
SEMANTEC [3]*	4m (30.6%)		✓	✓	✓		✓				✓
GENOME [5]+	1,260 (100%)		✓								
ANDRADAR [181]+	20,000 (N.A.)						✓	✓		✓	
ANDRUBIS [182]+	1m (N.A.)		✓		✓				✓	✓	

* company security reports; + academic security report

all crawled apps are reported as PHA, including adware. In this study, we rely on the popular AVTs from VIRUSTOTAL to detect the PHA for the further analysis and forensics. As the third-party testing lab AV-TEST reports [155], these popular AVTs exhibit a high accuracy of 95% above on average for *existing* PHA.

Different from the existing reports from IT-security companies, we analyze Android apps based on our own proposed representation model. Our analysis model aims to capture the possible malicious behaviors and other unique features of an individual app to be analyzed. The representation model depicts the basic information of Android apps (i.e., the 15 types of program features, see Section 6.3). For benignware and different families of PHAs, we build and compare their representation models. Based on these basic program features, we also apply associate rule mining techniques to mine some high-order features that are frequent item-sets of basic feature values (see Section 6.3.4). These high-order features show the a more meaningful high-level relationship of PHAs. Hence, we propose six dimensions of features on which the Android PHAs should be clustered and analyzed:

- *time* that shows the popularity of (a type of) PHA in a specific period;
- *market* that exhibits PHA may be affected by the geographic factors, users communities, and so on;
- *author* that shows the creator of Android PHA;
- *platform* that shows PHA may take OS vulnerability as each Android OS version has different security policies;

- *version* that shows how different versions of the same package affect the behaviors of apps and the results of detection of AVTs;
- *family* that shows their category and the attack of the PHA.

To the best of our knowledge, this is a first attempt to conduct a trend analysis on such a large scale of Android PHAs. As shown in Table 6.1, previous industrial security reports [3, 14, 180, 183] usually provide the statistic and spatial analysis of PHA, and focus the threats and attack matters of new emerging PHA. Security companies focus on PHA, PHA family and marketplaces analysis. However, they usually seldom investigate the authorship of PHA, the relationship of PHA to vulnerability, and the security mechanism of marketplaces. Previous academic studies only focus on individual apps, e.g., [20, 56, 182, 184] can detect malicious apps in an efficient and effective way, [185] is to analyze the permission and risks of Android apps, [186, 187] analyze the risk inside Android Ads libraries. In this thesis, we do not analyze one app individually. Instead, we analyze a large scale of PHA. To observe the popularity and trend, we explore the correlation and connection between different PHA families.

Due to the efficiency of our analysis approach, each app only requires less than 10s for feature extraction. After running our approach for months, we analyzed all the 4,267,178 Android apps and obtained the answers for the concerned questions. We summarize the following findings: 1) Complying with industrial reports, Android PHAs extensively hide in popular markets, even with a much higher percentage than it was reported. 2) PHAs are increasing at an exponential rate, and we observe several manners of automated generation. 3) Is Android OS becoming more secured? Yes. However, more PHAs emerge than before, as new variants due to piggyback attack and the new attack vectors for new vulnerability. 4) The existing markets are still weak at detecting PHA and a guess of the PHA detection mechanism of markets is provided. Overall, all of our findings facilitate the comprehension of Android PHA and its evolution, and thereby help to enhance and improve the existing Anti-malware tools. They influence all stakeholder such as users, developers, market administrators, and security companies, and are of great benefit to the whole Android ecosystem.

To sum up, we make the following contributions:

- We propose a representation model, which can depict the basic information and characteristics of Android app. We leverage representation model for the large scale app and PHA analysis.
- We propose 15 features to characterize Android app or PHA. We also apply frequent-itemset mining to mine those high-order features that are the frequent value sets of basic features. Further, we can measure PHA similarity in terms of the six dimensions, e.g., time and market.
- We report the findings of our large-scale app analysis. We shed light on the authorship and auto-generation of PHA, the relationship of PHA and vulnerability, the correlation of PHA families, and the detection mechanism of markets.
- Based on the analysis model and dimensions, we have investigated 4,267,178 Android apps, including 23.5% of them as PHA inside. We publish this huge PHA data set for research purpose.

6.2 Background and Questions

There are hundreds of Android markets, including the official website GOOGLEPLAY that provides millions of Android apps for mobile users [72]. Many efforts have been invested to analyze such a huge amount of apps by both academia and industry. This area covers security risk assessment [188–190], privacy leakage detection [21, 22, 40], PHA detection [20, 44, 57, 191], and vulnerability discovery [192, 193]. However, these works mainly analyze the apps, individually, while lack the perceiving of correlation amongst these apps collectively. In this section, we state the background of this work and present the questions to be answered in this thesis.

6.2.1 Android Malware

The first Android PHA was detected at 2010 [194] as a Trojan, which opened a backdoor for attackers to access the victims' resources illegitimately. With the evolution

and derivatization of a long period of time, Android PHA presents a variety of attack targets, attack techniques, and evasive techniques to prevent the detection. According to the security report in 2016 [3], there emerge 121 types of new PHA families, 9,433 PHA samples active in our daily life since 2013. In this work, we use VIRUSTOTAL[1] as the oracle to determine whether an app is malicious or not. VIRUSTOTAL integrates 54 antivirus products and displays the output of these antivirus products. The antivirus product will provide a label for the tested app if it is PHA, for example, one of the antivirus products, F-Secure identifies one app in our data set as "Trojan:Android/Vdloader.A", AVG reports that it is "Android/G2P.H.80CDE7D8828F", and BitDefender returns the result of "Android.Adware.Youmi.A". For simplicity, we consider all *potentially harmful application* as PHA due to their potentially malicious intention.

A benign app may be turned into PHA by: 1) third-party attackers to disseminate PHA, which is called *piggybacked* PHA [63]. This kind of PHA is very prevailing and can spread very rapidly in Android marketplaces [56]; 2) its owner to deliberately integrate malicious code, which is called *update attack* [5]. Some apps may disguise themselves as normal apps in the primary versions, while carry malicious code in their subsequent versions. The update attack is to bypass the strict inspection on the firstly-uploaded apps, as the scanning of the afterward versions of the app may not be strict.

6.2.1.1 Malware Family

Malware family is defined as the group of PHA samples with similar malicious behaviors. As shown by the two popular PHA data sets GENOME [5] and DREBIN [20], variants in the same family usually have the same author, the similar attack targets and techniques, and similar or even identical malicious code. For example, the infamous PHA family DROIDKUNGFU is known to steal sensitive information, such as IMEI code and phone model, and exploit vulnerabilities to elevate its privilege. In its later versions, i.e., newly evolved variant, it leverages advanced techniques (e.g., encryption of payload) to bypass the detection and exploit more vulnerabilities. The study on PHA

[1] http://www.virustotal.com

family can be used to track the authors of PHA, summarize the malicious code, and identify the evolved variants.

6.2.1.2 Normalization of Malware Family Names

Every antivirus software vendor (AVT) maintains a list of PHA families. They extract patterns or features to represent PHA families, and uses them to identify the suspicious apps. For one PHA, different AVTs may adopt different family naming methods. For the ease of the analysis, we perform the normalization on the names of PHA families. In this work, we employ AVCLASS [61] to infer a normalized family name for PHA. AVCLASS uses a probabilistic model to identify the most likely family name for PHA, and gives an unknown tag for PHA which cannot be successfully classified.

6.2.2 Questions for Understanding PHAs

6.2.2.1 Common Questions (CQ)

We summarize the common interesting questions that are answered by the industrial reports [3, 14, 178, 180] and academic studies [181, 182]. In this thesis, we aim to answer these questions in a comprehensive way via the study on the large dataset that we have.

1 App Analysis. Android system uses package name as the identifier to identify one unique app. Generally, the apps with the same package name are similar products developed by the same author. For each app, there exist a number of variants with minor modifications or of different versions — in our data set, 2.61 variants for one app on average. Some variants are PHAs, as they are injected with malicious or unwanted code. Among variants of one app, we omit the variants of regular update or defect repair, but focus on the variants that become corrupted (PHA). App Analysis helps answer these questions:

- **CQ1.1 How and when is the app changed?** We attempt to understand the attack features of PHA that are turned from benign apps, and malicious behaviors in the alternation which can be harmful to users. Besides, among different variants of an app, we want to investigate the version factor that relates to piggyback or update attacks.

- **CQ1.2 What are the popular attacks? How do different attacks/families constitute?** We attempt to unveil the details of the changes that lead apps into PHA. In addition, the understanding of the attacks contained in the modifications can benefit the security communities of knowing the trend of popular attacks, and taking the corresponding measures to defense against these attacks.

2 Malware Family Analysis. As a PHA family is formed based on the common malicious behaviors, it is helpful to know the evolution of characteristics of PHA variants in the same family, and other metrics on Android PHA. From this study, we aim to summarize the usage patterns for permissions or sensitive APIs in PHA variants during the evolution, and how they spread to understand their life cycle.

- **CQ2.1 How does PHA number increase?** The study on how the model spreads can be used to propose the corresponding countermeasures against attacks.

- **CQ2.2 How does PHA family evolve?** We build an abstract and semantic model to represent the characteristics of Android PHA, including the invocation of sensitive APIs, the requested permissions, and so on. The characteristics vary from different variants, because of either a shift in attack targets, or attack techniques, or even the development of PHA detection techniques. The evolution analysis can update our knowledge of PHA, and facilitate the future PHA detection, and system enhancement.

- **CQ2.3 How do different families relate (co-evolve)?** Differences in characteristics of PHA variants can reveal the variety of malicious behaviors. In addition, it is inspiring to learn the commonalities of malicious characteristics between these PHA families, and the relationship in between.

3 Market Analysis. Android marketplaces, GOOGLEPLAY and other third-party marketplaces, provide millions of apps for global users. As an open framework, they allow any companies, organizations, even individual developers to upload their apps. However, the examination and inspections of apps cannot guarantee all apps are harmless.

- **CQ3.1 How does PHA distribute and spread among different markets?** The distribution of PHA in markets reveals attackers' preferences towards specified users. We attempt to perform a reginal analysis to study how the PHA is distributed across Android markets.
- **CQ3.2 What are the possible detection mechanisms of each market?** Generally, one app can be uploaded into multiple markets. Due to different inspection mechanisms, one app can be removed from the shelves if it is detected as PHA. Hence, the venue of Android PHA infers the risks of the markets and the safety of the market users.

6.2.2.2 Our Questions (OQ)

In this thesis, we adopt two more types of analysis to answer the questions that are usually not covered by the existing reports.

4 Authorship Analysis. Knowledge of authors of Android PHA is a great leverage for security experts to identify the behaviors of PHA against perpetrators. It is straightforward and efficient to assess the security risk of apps based on the probability of their authors being perpetrators. The study of distribution of PHA with the same author can imply the preferences and thereby facilitate the protection. In addition, the study of the behavioral patterns of PHA authors can facilitate the understanding of intentions and regularities, which helps to improve our protection mechanisms.

- **OQ1.1 Who turns a benign app into PHA?** As benignware can be turned into PHA by piggyback or update attack, it is interesting to know who are authors and whether the piggyback or update attacks are conducted by different attackers?

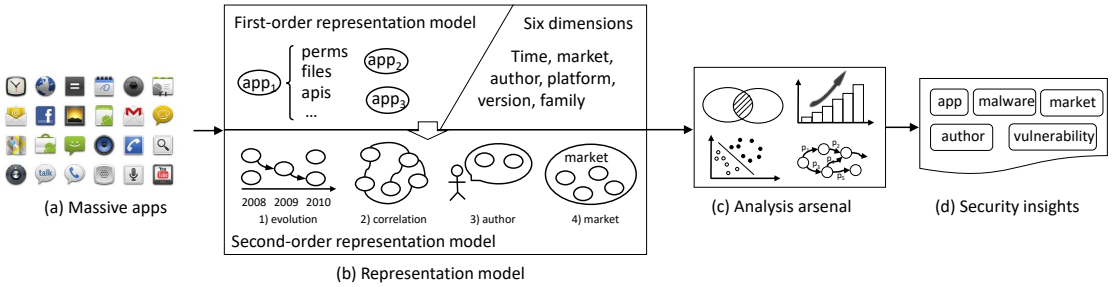

FIGURE 6.1: The overview of the approach

- **OQ1.2 What are the behavioral patterns of PHA authors creating PHA?** The technique of auto code generation greatly facilitate the emergence of PHA and its variants. We attempt to identify the employed auto-code generation techniques, which can give a very useful hints to fast identify PHA variants.

5 Vulnerability Analysis. Since the first release of Android in Sep. 2008, it experiences 24 times of major upgrade, and hundreds of security enhancements and patching. Meanwhile, the vulnerabilities in Android are exposed to the public from time to time, which can be exploited by PHA to launch attacks. GODLESS [195] is one kind of PHA to root Android devices, which is reported by TRENDMICRO in June 2016. It exploits multiple vulnerabilities in Android 5.1 (Lollipop) or earlier to elevate its privilege. Fortunately, AOSP (Android Open Source Project), led by Google, always releases the patches for exposed vulnerabilities and enhances the security of system, and thereby avoids the similar attacks in the patched version. Apparently, Android PHA is affected by these two types of events, vulnerability disclosure and system upgrade. Hence, we want to understand the correlation of PHA and vulnerabilities in Android OS.

- **OQ2 How does the PHA popularity relate to the new vulnerability?** PHA may take advantage of the common weakness (CWE) or vulnerability (CVE). We conduct the statistic correlation analysis for the PHA appearance and vulnerability discovery. We also manually analyze some PHA family for scrutiny of the correlation.

Last, we leverage the data mining technique (i.e., association rule mining) to find and visualize the potential correlation of PHA on different dimensions.

6.3 Analysis Approach

In this section, we introduce the representation model for Android apps, which can help to distinguish PHA from benign apps and tell differences between PHA families. Based on the representation model, we cluster the apps on different dimensions and conduct the further detailed analysis.

6.3.1 Approach Overview

Fig. 6.1 illustrates the overall steps of our analysis approach. Taking as input millions of apps in Fig. 6.1(a), our approach extracts the corresponding labels (e.g., the apk information, the detection results of AVTs) and produces the conclusions and highlights as shown in Fig. 6.1(d) that respond to the problems in Section 6.2.2. Hence, our approach has the following pipeline processing:

- **Massive apps**. They are the input for the whole analysis, and collected from multiple sources and a long range of time (see Section 2.4).
- **Representation model**. We employ two-order representation model to depict the dataset. As shown in Fig. 6.1, the first-order representation model is extracted by static analysing Android apps, and to depict the characteristics of one single app. Combining with six dimensions (see Section 6.3.3), we build a second-order representation model to depict the characteristics of a group of apps. The details of the second-order representation model can be found at Section 6.3.4.
- **Analysis arsenal**. The constructed two-order representation models are passed to the analysis arsenal to perform different mining tasks. As discussed in Section 6.3.5, we conduct difference analysis to study why one app becomes PHA, machine learning to learn the common characteristics of PHA during the process of evolution.

- **Security insights**. Based on the analysis arsenal, we have drawn many insightful conclusions from five views which is discussed in Section 6.2.2. These conclusions, which are very practical and accurate, can benefit both the academia and industry.

6.3.2 First-order Representation Model

To depict the characteristics of an Android app, we propose a first-order representation model as the metrics. The *first-order* model refers to the model of two types of atomic features—*identifiers* and *properties*. The *identifier* features are used to identify an Android app, while the *property* features are used to describe the characteristics of Android apps. The identifiers are listed as follows:

- **Sha256** is a hash value for the whole apk file. Due to collision resistance of sha256, it can be used to quickly determine identical files. Two apps with the same sha256 value are equivalent at the bit level. Thus, differencing analysis is not necessary for the replicates with the same Sha256.
- **Certificate**. An Android app is compelled to be digitally signed before the installation, and thereby the certificate can be used to identify the author of apps. The same certificate with multiple apps can implicate the same author, while different certificates cannot implicate multiple authors as one author can own several certificates.
- **Creation date** represents the date when the developer (or attacker) creates this app. It can benefit the study on the evolution and prorogation of PHA, originality identification, and so on.
- **Marketplace** denotes the venue where an app is downloaded. This property can facilitate the risk assessment on different Android marketplaces.
- **Package name** is a Java-language-style name for Android apps. It serves as a unique identifier for apps in one Android device. Any conflict of package name amongst installed apps on device is prevented by Android OS.

- **Version code** serves as an identifier to specify the version for a certain app. Due to the requirements of enhancement and improvement on functionalities and security, Android developers maintain their products in different versions. The version code can be combined with package name to specify an instance of product.

The properties includes the following:

- **Requested Permission**. Android apps have to request permissions to access the specific resources. For example, android.permission.SEND_SMS is a must to invoke the method sendTextMessage to send an SMS message. The permissions can reveal the resources the app intends to access, which are likely abused.
- **Sensitive API**. Apps access controlled resources by invoking APIs provided by Android. The invocation of sensitive APIs represents the real operations on these resources on Android. It can characterize the functionalities of an app, and thereby is an important criterion to determine the maliciousness.
- **Native Code**. Oftentimes, native code is used to interact with the C/C++ layer in Android [196]. These interactions are possibly conducting an attack to elevate the privilege on device. For example, CVE-2010-EASY can be exploited to root devices under the version 4.0.
- **Reflection**. Java reflection is widely used in Android apps for various purposes such as modular development [168], dynamic patching, and dynamic loading of payload or malicious payload [13]. The feature can depict the existence of dynamic invocations in apps, which raises a security risk.
- **Contained File**. Android PHA can hide malicious code into resources files under the folder "asset" or elsewhere. In this case, we can recognize the suspicious files especially when comparing the repackaged apps to the original one.
- **Component**. Android defines four components as building blocks to construct an app. The components which are *activity*, *service*, *broadcast receiver* and *content provider*, serve as the skeleton and reveal the basic functionalities of Android apps.

- **PHA Family**. If an app is detected as PHA, the PHA family is a label to illustrate the category that the app belongs to. Generally, PHA in the same family has a large portion of similar malicious behaviors. For example, variants in DROIDKUNGFU of GENOME all have malicious behaviors to steal sensitive information such as contacts.

- **Sensitive Strings**. Sensitive strings mean the strings used in an Android apps which are security-related. In this thesis, we consider the strings of URLs that are commonly used by Trojan apps, and phone numbers that are commonly used by financial apps, and file names that are used by ransomware and spyware.

- **Dependent Libraries**. Libraries should be explicitly claimed in the manifest file if an app wants to access some resources, especially the hardware resources. For example, an app has to claim the feature android.hardware.camera before using camera.

We employ a lightweight static analysis to extract the information from Android apps, and run the service VIRUSTOTAL to get the detection result. AVCLASS is used to normalize the family name of confirmed PHA.

6.3.3 Cluster Dimensions and Strategies for Apps

It is infeasible to conduct the analysis on the whole dataset. Therefore, we propose six dimensions to cluster apps based on our interests. The dimensions are as follows:

- **Time**. PHA created in a specific period of time can reveal the typical concern of attackers. We can conclude many insights on these apps with common targets.

- **Market**. Apps across different markets may be different with injection of code by the administrator of markets. We investigate the changes in different markets, and whether these changes can turn the app into PHA.

- **Author**. The certificate identifies a unique developer. Clustering apps with the same author can learn the behaviors of the developers from the prospective of forensics.

- **Platform**. Apps are designed to run on a specific platform, which is defined in the manifest file with *target_sdk*, *min_sdk*, and *max_sdk*. Clustering app in light of platform can facilitate the study, for example, the correlation between attacks and the platform, and the countermeasures taken in a specific platform.

- **Version**. The analysis in version dimension, which is targets at the same app of different versions. There exist several cases with this clustering strategy. For example, the app is benign at first, while the app becomes PHA in a specific version. It can be used to investigate the reason of the changes. The app is determined as PHA, while becomes benign app since a specific version. By investigating the differences between the different versions, we can determine whether the evasion techniques are used to bypass the detection or the reason to cause the harmfulness of app.

- **Family**. The common and different parts of the apps in the same PHA family. The common parts may be the essential parts of malicious behaviors which can be used to identify this class of PHA, and the different parts may show the variety of malicious behavior or other non-malicious code.

These dimensions guide the clustering process during the large-scale analysis. Note that they are not mutually exclusive, and can be combined to address different problems.

6.3.4 Second-order Representation Model

The first-order representation model just characters a single Android app. In order to reveal the characteristics of a certain types of apps, we propose a second-order representation model to combine first-order features according to the clustering dimensions. The second-order features are extracted from our data set via the data mining techniques, i.e.,

associate rule mining. In our problem domain, apps are analogous to the transactions in the data mining domain, and first-order features are analogous to the items.

Hence, give a large data set of apps (benign and malicious) with their labelled first-order features, it is desirable to mine some frequent occurrences of first-order feature values in terms of a given analysis dimension. This can be formalized as below, and resolved with these methods [197].

- $F = \{f_1, f_2, ...f_m\}$, which denotes values of first-order features (set of items);
- $A = \{a_1, a_2, ..a_n\}$, which denotes the apps (transactions). Here, a_i refers to a value vector of first-order features;
- $HF = \{f_j, ..., f_k\}$, which denotes to a high-order feature (item-set), where features $\{f_j, ..., f_k\}$ frequently occur together with a significant confident and support in all apps.

Evolution feature. The feature depicts the changes among apps along with time or version upgrade. By introducing time and version information into the model, we can draw useful conclusions such as typical features that lead apps into PHA, and how advanced features or techniques are employed during the process of PHA evolution. For example, we identify the essential features that lead the app to become the PHA "fakeinst". For a PHA family, we mine some high-order features of the requested permissions, creation date and version code. Thus, we can learn the trend for a specific feature.

Correlation feature. The correlation features between apps reveal the distance between apps, that inspire the security community to conduct a mutual defense to a similar attack based on existing knowledge. For example, the high-order features of PHA family name, dependent libraries, request permissions and sensitive APIs can effectively show the relationship and similarity of these families. Thus, these features can also facilitate the identification of the source of PHA, and the propagation direction in the future.

Author behavior feature. From the prospective of author, we can identify the specific behavioral patterns of a certain PHA author, such as required permissions, code-written patterns, and attack targets. For example, we identify the high-order features of authors,

certificates and markets. These high-order features show which authors love to use the same certificate for PHA submission to certain markets.

Market defense feature. We introduce the market and PHA family information into the representation model, and summarize the status quo of Android PHA in a specific market. The features can reveal the distinguished characteristics of app across Android markets, and the resistance capabilities against different Android PHA.

6.3.5 Analysis Arsenal

In the fourth step, we leverage multiple learning techniques to solve the problems defined in Section 6.2.2. According to our targets, we have employed the following learning techniques:

Differencing Analysis. We perform the differencing analysis to obtain the commonalities and differences between two apps. The differences in between can be employed to identify the unique and typical characteristics of Android PHA and benign apps. The differences between one PHA sample and its variants can facilitate the study of PHA evolution, thereby helping to improve the existing PHA detection, and better understand the trend of PHA in future.

Statistic Inference. Statistical inference is the process of deducing properties relying on the distribution of analyzed data. We employ it to infer the predominant characteristics and evolution of PHA (see Section 6.4.2.3) from the statistical and probabilistic models built on apps. In addition, we infer several behavioral patterns of PHA authors (see Section 6.4.4) by using statistic inference.

Correlation Analysis. Correlation analysis is an approach to compute the relevance between multiple variables. In this thesis, we identify the correlations between PHA families in characteristics in Section 6.4.2.3. By computing the relevance between the number of Android PHA and the security-related events in Android world, we conclude several highlights for potential relevance in Section 6.4.5.

Machine Learning Clustering Analysis. Machine learning is commonly used to learn specified knowledge from big data. In this thesis, we leverage machine learning to learn the relationship between PHA in Section 6.4.2.3. By clustering PHA based on their properties, we show a detailed and compound connection existing in PHA.

6.4 Analysis Results & Comprehension

Our analysis approach is implemented in approximately 4,153 lines of Shell and Python. All the data processing and analysis are conducted on a Ubuntu 14.04 server with Intel Xeon E5-2699 and 160G Memory.

Research Questions. We conduct our analysis to answer those questions in Section 6.2.2. For these CQs that mostly have been discussed by existing reports, we compare our findings with theirs. For the OQs, we focus on reporting the unique findings.

6.4.1 Answer to CQ1 — App Analysis

Android apps may have different versions within the marketplace and different variants across marketplaces. In this experiment, we aim at the apps with same package name and version code, but different hash values and maliciousness. It is non-trivial to analyze the differences between these variants. Some variants of the same app may be PHA due to piggyback or update attack.

Based on the clustering dimensions of package name and version code, we obtain 27,560 groups of apps (71,888 apps in total) that have identical package name and version code. Hence, on average, one app has 2.61 variants across 28 marketplaces and 4 PHA collections. To understand the corruption, we analyze each app group that has at least one PHA.

Analysis procedure. We carry out two types of analysis on these selected apps: 1) compute the differences between apps (see Section 6.3.5), and identify the differences that turn a benign app into PHA; 2) we study app variants along the time line and identify the versions where the corruption happens.

Summary for CQ1.1: when and how the apps get corrupted?

- Apps seem to turn corrupted at the early stage of its evolution. On average, the corruption happens after 1.3 version iteration, and 21% of corrupted apps may become benign after 3.4 version iteration on average.
- *Most PHA variants are of piggyback attack from third-party attackers*. There are 21,037 (76.3%) variants that have different certificates with original authors. Hence, the variants are most likely tampered intentionally by perpetrators, not by the original authors.
- *Malicious code is widely and frequently reused for corruption*. In order to fast disseminate malicious code and launch attacks, well-crafted code is piggybacked into previously prestigious apps.
- *The injected code focuses on ads and native code*. In these 27,560 groups of variants, there are 60% variants which are injected with ads to make profits, and 10% which are put into native code. However, native code is commonly used to exploit vulnerabilities for privilege escalation on Android. Hence, native code can facilitate any attack.

Summary for CQ1.2: the popular attacks and the percents

- *Privacy leakage is still the most common target*. Although *ransomware* and *financial charge* have become the new weapon of attackers in these years [3], they tend to cause a huge damage or loss to users. Due to their explicit behaviors, they are removed once detected in general. As shown in Table 6.2, *privacy leakage* still takes a lead in Android PHA (14 out of 20 families with privacy leakage). This kind of attacks are walking along the boundary of PHA and benignware. It lacks a universal criteria to determine whether one app with privacy leakage is PHA or not, because some apps have the terms to officially collect the privacy for improving service quality. Thus, the works [22, 23, 40] only expose the risk of privacy leakage, but not decide the maliciousness.

Status of App Corruption. The subject of app corruption is the group of app variants which have the same package name but different detection results. It can unveil the root cause that lead apps to be malicious. We first conduct difference analysis to these app variants, and identify the differences, and then employ machine learning, associate rule mining exactly, to extract and conclude the characteristics of differences that lead to corruption as follows:

- Most of corruptions into adware (81%) are caused by original authors. Along with the functionality upgrade of app, the app author intends to integrate advertisement into app to make profits. Thus, there are a large portion (78%) of apps that introduce ads libraries and become PHA.
- Benign apps and corrupted apps can be distinguished by our features proposed in Section 6.3.4. For example, the features of PHA family "fakeinst" can be depicted with the requested permissions android.permission.SEND_SMS and android.permission.RECEIVED_SMS, a broadcast receiver thats can receive incoming SMS messages and make a response by sending out SMS messages.

6.4.2 Answer to CQ2 — PHA and PHA family

In this study, we intend to understand the increase rate of PHAs, the characteristics of a certain families of PHAs, and the correlation of different families of PHAs. Therefore, we group all PHAs on the dimension of family.

6.4.2.1 PHA increase rate

Modeling the PHA increase and spreading is a non-trivial topic, as there might be many factors and constrains to be considered. Insofar, we just use polynomial functions to approximate the curve of the increased PHA number along the time line, not to derive a theoretic model from these numbers. Interested readers can refer to [198] for regression analysis on the number of certain PHA families.

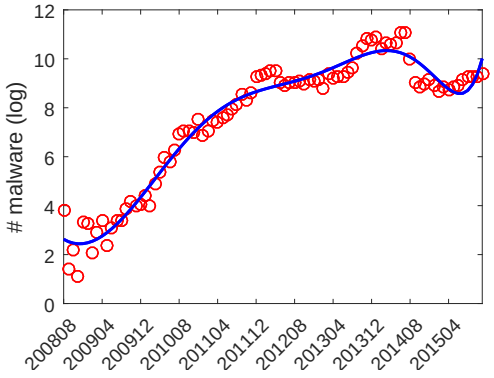

FIGURE 6.2: Growth curve for # PHA

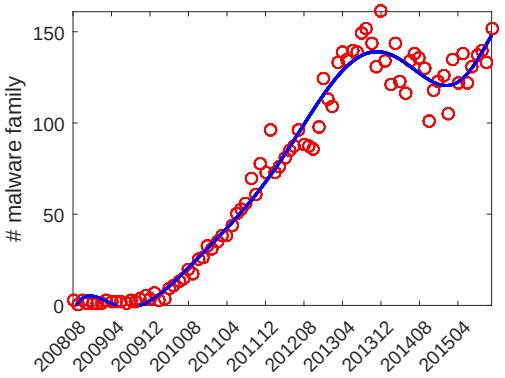

FIGURE 6.3: Growth curve for # family

Summary for CQ2.1: the increase rate of PHA. Fig. 6.2 shows the number of increased Android PHA by month, and the curve suited to the logarithmic number of increased PHA. Fig. 6.3 shows that the number of increased PHA families, and the curve suited to the number of increased families. In general, the PHAs and new families are keeping emerging at an exponential rate. One notable observation is that there is a clear drop of the increase (from e^{10} at 201404 to e^{9} at 201504), and then an outbreak after that. This observation is consistent with the MCAFEE mobile threat report 2016 [14] — before 2015 Q1, Android PHAs increase by $2 * 10^6$ per month; since 2015 Q2, new PHAs outbreak at the rate of 10^7 per month. Although we investigate only 28 markets that are less than hundreds of markets McFee investigated, the similar observations are obtained.

6.4.2.2 Evolution of PHA families

In this section, we conduct an investigation on the evolution of PHA families along the time since Android is unveiled in 2008. As shown in Fig. 6.4, we divide the history of Android PHA into three phases:

1. Age of chaos (2008∼2009), when there is little knowledge about Android PHA, and people are not well aware of the damages caused by Android PHA. In this phase, we find that a large portion of PHAs compromise the SMS related functions. For example, PHA families *smsreg*, *autosms*, and *smsagent* send sensitive information on device via SMS messages; PHA families *fakeinst* and *plankton* send SMS message to a specific

premium rate number to subscribe premium services; PHA family *gappusin* is a trojan that uses SMS message to communicate with the C&C server; PHA family *bgserv* can be controlled by remote attackers to send SMS messages, and do malicious actions such as PHA promotion. In addition, we find that PHA is already enclosed with multiple adware in this phase. For example, we find that PHA family *gappusin* contains the adware *airpush*, *umeng*, *kuguo*.

2. PHA explosion (2010∼2013), when there is a booming increase of Android PHA. The PHA named "Trojan-SMS.AndroidOS.FakePlayer.a" is the first Android PHA identified by Kaspersky in 2010. This PHA attempts to send messages to premium rate number in Russia, and cause financial loss of users. The identification of the PHA drew people's attention to Android PHA, and shed light on the significance of the protection from Android PHA. Meanwhile, together with the rapid development of Android and its ecosystem, a tremendous number of PHA are created to make profits for cybercriminals. Second, three automatic techniques aggravate the explosion of Android PHA (see details in Section 6.4.4). Last, many advertisers swarm into Android, and distribute adware to make profits. Adware, which takes 78.1% of malare in this phase, has caused varying degrees of damage to users. All these factors lead to a rapid increase of Android PHA by 414%, 185% and 114% from 2011 to 2013.

3. Age of vulnerability (2014∼), when the PHA prefers exploiting vulnerabilities to launch attacks. These three years are the time that has the most growth of Android vulnerabilities. As shown in Fig. 6.4, there are 125 and 462 vulnerabilities found in 2015 and 2016, respectively. The famous vulnerabilities exist in stagefright (a media server), Adobe Flash Player, and OpenSSL. The exploits of vulnerabilities, which is usually written in native code, raise the difficulty of being detected. According to our investigation of the exploits of vulnerability "stagefright", there are totally 41,664 PHA exploiting stagefright, but only 8,583 (20.6%) PHA are detected. In addition, we observe that thousands of Android markets facilitate the spread of Android PHA these years. The illustrative figure in Fig. 6.4 shows a simple spread model of Android PHA. Two markets may have a close relationship that they may share or acquire apps from each other. For example, the market APP20 is confirmed to crawl popular apps from GOOGLEPLAY. So one market that is infected by a certain kind of PHA may just pull

apps from other markets without security inspection. In addition, the PHA can be directly uploaded into a certain market by attackers. The information exchange between different markets speed up the spread of Android PHA, which would be disccussed in detail at Section 6.4.3.

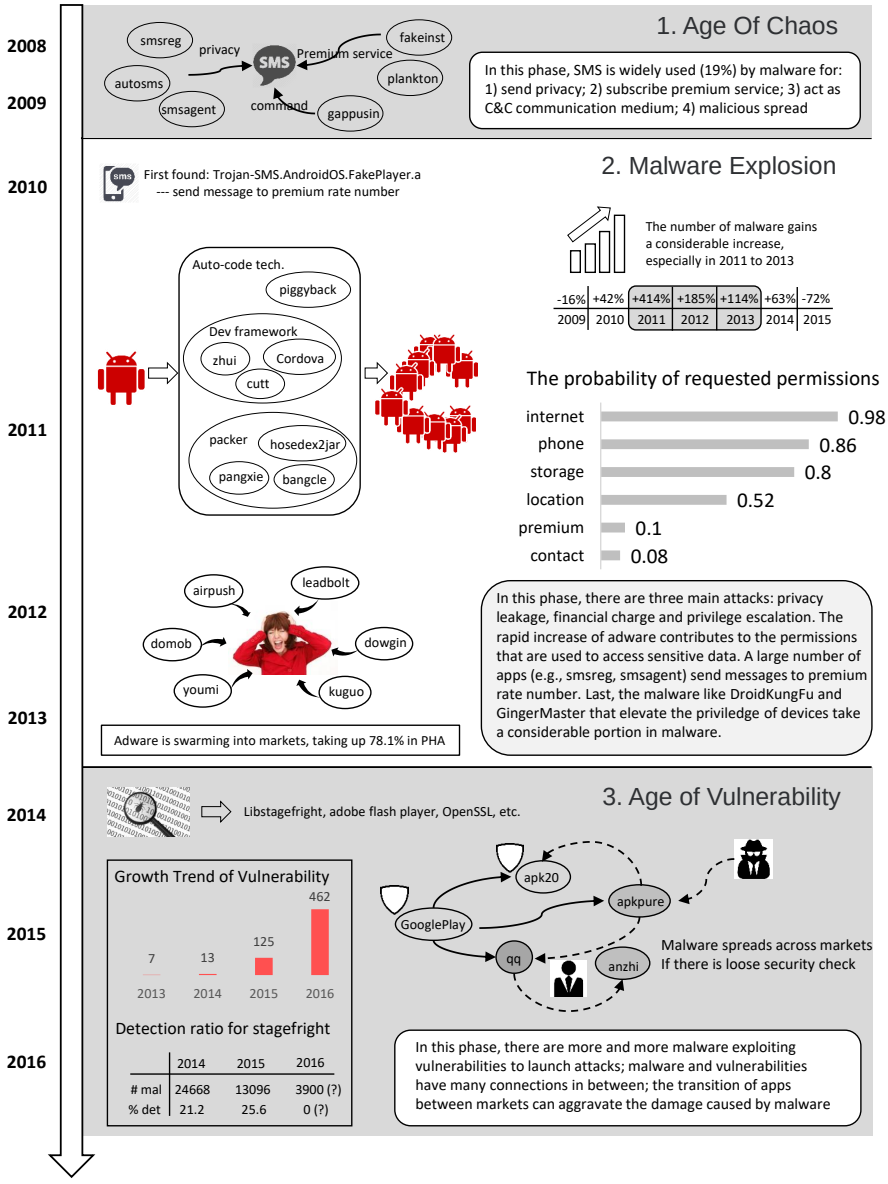

FIGURE 6.4: The brief history of Android PHA

6.4.2.3 Relation between PHA families

Among the 1,400 Android PHA families in our data set, each of them has its own characteristics (e.g., the attack targets, the writer, and the resources accessed from Android

TABLE 6.2: Top 20 PHA families in our data set without the consideration of adware. We summarize the triggers of PHA [1]-*main* means the startup of PHA; *broadcast* refers to broadcast messages of system such as an event of incoming SMS message; *observer* refers to the change to content provider. The attack targets mainly come from [3, 5], of which *control* refers that the device is controlled by remote attackers or by local PHA to conduct malicious operations. For example, PHA *ginmaster* downloads malicious payload from remote server, PHA *opfake* can delete SMS messages on device, and PHA *spyagent* can block the operation of uninstallation.

Family	**Trigger**			**Attack Target**						# Samples
	main	broadcast	observer	privilege	control	finance	privacy	disguise	ransom	
gappusin	✓				✓		✓			10,331
secapk	✓							✓		10,169
smsreg			✓				✓			11,472
plankton	✓				✓	✓	✓			2,909
anydown	✓								✓	1,566
fakeinst	✓					✓				986
skymobi	✓						✓			933
uapush	✓					✓	✓			843
deng	✓					✓				777
ginmaster	✓			✓	✓		✓			631
opfake	✓				✓	✓				622
apptrack		✓					✓			546
smsagent	✓						✓			477
tekwon		✓		✓						330
spyagent	✓						✓			290
fobus		✓			✓	✓	✓			266
mseg	✓	✓		✓			✓			222
stopsms	✓			✓			✓			209
droidkungfu		✓		✓	✓		✓			183
goldentouch	✓						✓			161
Total	15	5	1	5	5	6	14	1	1	43,923

system). Current researchers are eager to detect PHA and know what the PHA can do. In this experiment, we build a probabilistic model for each family, and compute the similarity of probabilistic models of these families. The result can further facilitate family classification and explain the included malicious behaviors.

There are 1,004,550 PHA variants in our data set. Based on the clustering dimension of PHA family, we classify 200,373 (20.0%) PHAs into 1,400 families using AVCLASS.

Analysis procedure. In order to analyze the differences in characteristics of PHA families, we first select nine kinds of property elements to represent one PHA sample—*declared permissions, requested permission, libraries, activities, services, providers,*

receivers, contained files, and *invoked APIs*. Let PHA m be denoted as a set of property elements: $\{ins \mid ins \in props\}$, where ins is an instance of nine property elements. Then, we build a probabilistic model to represent each family based on these property elements of PHA samples. Assuming F is a PHA family, its probabilistic model can be denoted as a vector of the probabilities of property elements being relevant to F:

$$\mathscr{M}(F) = (\mathscr{P}(ins_1 \mid F), ..., \mathscr{P}(ins_9 \mid F))$$

where $P(ins_i \mid F) = \sum_{m \in F} P(ins_i \mid m) * P(m \mid F) = \frac{|\{m \in F | ins_i \in m\}|}{|F|}$

$P(ins_1 \mid F)$ reflects the probability of the existence of instance ins_1 in family F. In order to compute the similarity between two PHA families F_1 and F_2, we employ *cosine similarity* [199] on the probabilistic models to measure their similarity: $cosine(\mathscr{M}(F_1), \mathscr{M}(F_2))$.

Fig. 6.5 shows the correlations between PHA families at some aspects. We select the top 100 PHA families in our data, and identify the pairs of families whose similarity is larger than 0.5 (the baseline for classification). In addition, we select the top 20 PHA families which are not adware, and identify the pairs of families if any of property elements are similar. In addition, we have summarized these 20 PHA families in Table 6.2 from two aspects: *trigger* (how to trigger the execution of malicious code) and *attack target* (what are the targets of attacks). As shown in the right upper corner of Fig. 6.5, only 10 families have a cosine similarity of probabilistic models larger than 0.5. Hence, most families are quite different, due to their significantly different patterns of property elements.

If considering only *requested permissions* (req_perms) and *invoked libs* ($libs$), in the left part of Fig. 6.5, many families show a high extent of similarity. For example, *smsreg* requires highly similar permissions (0.94) with *gappusin*, and invokes the exactly same libs (0.99).

Summary for CQ2.3: the correlation of PHA families.

- *Based on all basic property elements, PHA families show distinct differences.*

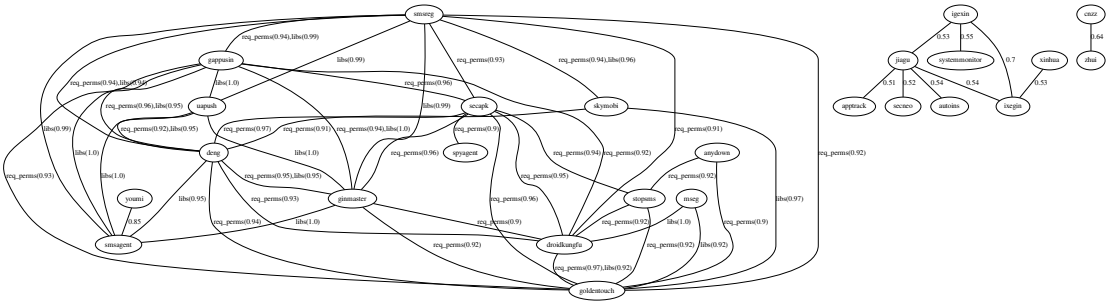

FIGURE 6.5: The correlation between PHA families. The node "◯" denotes one PHA family. If two nodes have edges in between, they are considered to be similar in some aspects. The labels on edge denote either the similarity of two nodes, or the similarity in some aspects. For example, the edge between PHA *droidkungfu* and PHA *mseg* indicates that these two PHA families leverages the same libraries.

There are only 10 pairs of PHA families whose similarity is above 0.5 (the borderline). Considering that there are 979,300 ($C_n^2, n = 1400$) pairs of PHA families in total, the percentage is extremely small and negligible. Hence, that explains why family based classification based on program features can be effectively fit to the modern Android PHAs. As no two families alike, classification techniques can be applied to identify PHA, and its potentially related family in an accurate and efficient manner.

- *Requested permissions and invoked libs are not unique to PHA families.* As shown in the left part of Fig. 6.5, these two features are highly overlapping among families, and too coarse-grained to characterize Android PHA begaviors. Moreover, permissions are often unduely used in Android apps [200–202]. Especially in Android 6, all the permissions need to be acquired at the installation time. Hence, detection based on these two features might not be effective.

6.4.3 Answer to CQ3 — Marketplaces Analysis

Apps are collected from 28 different marketplaces in a variety of different languages and regions. We summarize the proportion of PHA in each marketplace, investigate the markets' susceptibility to PHA, and the defence capabilities with regards to the potentially employed AV tools. We attempt to show to market administrators and stakeholder a reputation model of Android markets, which make them better improve the security of Android markets or be aware of potential downloaded PHAs.

We consider all 1,004,550 PHA samples in the dataset, and cluster them based on the dimension of marketplace. Since we have apps (including PHAs) submission time on various markets and app replicate ratio among markets according to, we can conduct the app migration (or PHA spread) analysis.

Before answer CQ3, we introduce the concept of *corruption probability (CP)* for an app, which denotes the probability of an app being corrupted among its variants. Give n versions of an app a, denoted as $a = \{\alpha_1, ..., \alpha_n\}$, its CP is defined as

$$p_c(a) = k/n$$

where k refers to the number of corrupted versions of app a.

Based on this, we introduce the concept of *mean corruption probability (MCP)* for a market M. Give the set of apps $A = \{a_1, ..., a_m\}$ on M, its MCP is defined as

$$\overline{p_c}(M) = \sum_{a_i \in A} p_c(a_i) / \mid A \mid$$

MCP is an important indicator that shows how risky on average a benign app could be corrupted on a given market.

Summary for CQ3.1:PHA distribution analysis across markets.

For each market, we compute MCP and show the top five markets of biggest/smallest MCP in Table 6.3. The five markets with a higher risk contain 173,583 apps while the other five markets contain 28,061 apps, only 16.2% of the former one. In addition, we sketch Fig. 6.6 to denote the migration of apps across different markets. The node represents the market, and the edge represents the app replicate ratio from the ending node to the beginning node. For example, the node *Wangyi* has an edge pointing to the node *mob*, and the edge has a label 1.0. This means the apps in *mob* are 100% contained in *Wangyi*. To sum up, many small markets copied apps from those big markets where the apps originate. *The way Apps and PHAs spread among markets should be effectively modelled by the epidemic model* [181]. Our preliminary study on PHA spread model can also be found at [198].

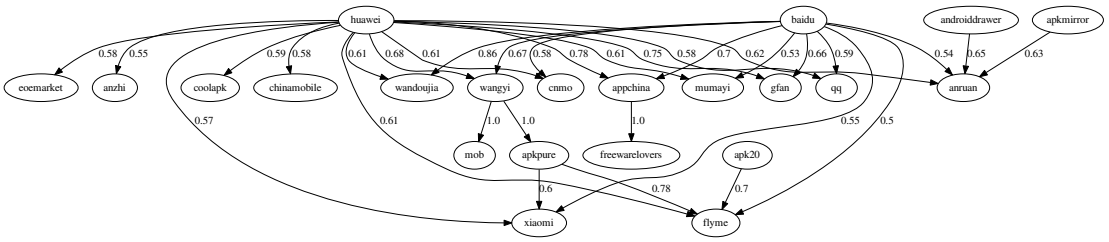

FIGURE 6.6: The corruption of apps across markets. Thce node "◯" denotes a market; The edge "Ⓐ $\xrightarrow{p}$ Ⓑ" denotes that if two apps, with same package name and version code, exist in market A and B, respectively. The app in market B has a higher probability p of being PHA. For ease of view, we only draw edges, of which the probability is larger than 0.5.

TABLE 6.3: Mean corruption probability (MCP) for differnt market. We list top 5 markets with the biggest MCP and top 5 with the smallest MCP.

	appchina	wandoujia	gfan	chinamobile	qq
MCP	72.7%	68.8%	68.5%	60.6%	59.7%
	freewarelovers	baidu	androiddrawer	fdroid	appsapk
MCP	8.0%	8.0%	6.5%	2.2%	0%

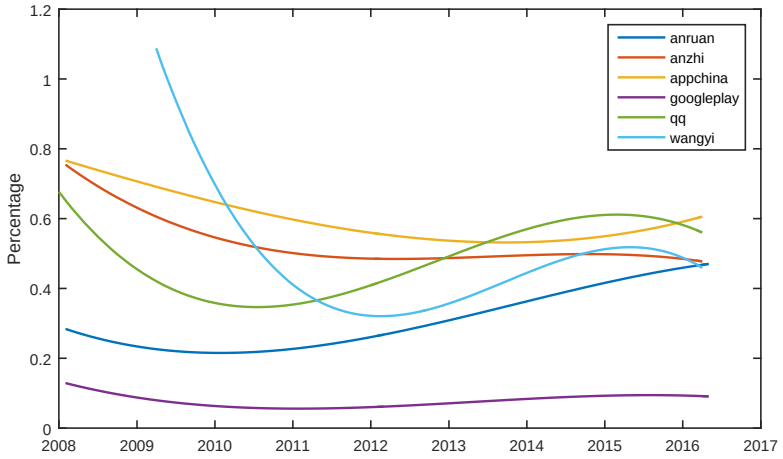

FIGURE 6.7: Top 5 marketplaces that have the largest ratio of detected PHA in the total apps (i.e., PHA ratio). The x-axis denotes the time since the find of the first PHA in the marketplaces, and the y-axis denotes the PHA ratio.

Summary for CQ3.2: detection mechanisms of different markets.

Fig. 6.7 shows the top 5 marketplaces that contain the largest percentage of detected PHA in total apps. Note that GOOGLEPLAY is included in Fig. 6.7 as baseline for comparison. As shown in Fig. 6.7, GOOGLEPLAY has a relatively low PHA ratio (9.5%), which is a bit higher than that (2%) from Google's security report [178]. After

TABLE 6.4: The selected marketplaces and their status quos of susceptibility of Android PHA. The susceptibility contains 5 top PHA families, of which the markets are susceptible, as well as its susceptibility rate.

Market	Susceptibility of PHA(%)					Attack Type
	MF1	MF2	MF3	MF4	MF5	
anruan	dowgin (29.3%)	wooboo (2.0%)	youmi (1.6%)	secapk (0.9%)	smsreg (0.8%)	privacy, adware
anzhi	dowgin (14.3%)	kuguo (10.6%)	secapk (5.5%)	domob (4.2%)	smsreg (2.6%)	privacy, adware
appchina	gappusin (9.4%)	smsreg (7.4%)	secapk (3.4%)	adwo (3.2%)	igexin (3.1%)	privacy, control, adware
qq	kuguo (17.7%)	gappusin (5.7%)	dowgin (3.6%)	secapk (3.6%)	wkload (3.2%)	privacy, control, disguise, adware
wangyi	jiagu (4.6%)	domob (4.6%)	secapk (2.7%)	smspay (2.5%)	smsreg (2.5%)	disguise, adware, privacy
googleplay	revmob (8.0%)	airpush (7.9%)	leadbolt (4.7%)	anydown (1.5%)	appsgeyser (1.4%)	privacy, adware

TABLE 6.5: The selected marketplace and suggested combinations of AV tools for these marketplaces. The column "Detection Rate" denotes the detection rate of PHA if the market employs the suggested combination to detect PHA.

Market	Detection Rate	# of AV tools	AVs
anruan	75%	4	AegisLab, AVG, F-Secure, and IKarus
anzhi	90%	6	AVG, NANO-Antivirus, Cyren, Ikarus, Avira, AhnLab-V3
appchina	90%	4	ESET-NOD32, NANO-Antivirus, AVG, Bkav
qq	90%	5	F-Prot, AVWare, Alibaba, Bkav
wangyi	90%	4	Bkavv, ESET-NOD32, Cyren, AVG
googleplay	70%	7	ESET-NOD32, NANO-Antivirus, Fortinet, Ikarus, Vipre, Antiy-AVL

the scrutiny, around 80% of PHAs in GOOGLEPLAY are reported as *Adware*. Since GOOGLEPLAY is launched in 2008, its examination to uploaded apps has experiences several revolutions [1, 203]. The combined verification techniques can greatly detect PHAs and block them from spread. On the contrary, third-party marketplaces, which may have a relatively loose inspection on uploaded apps, have a considerate PHA ratio. These five markets are from China, where users cannot directly access the service of GOOGLEPLAY. These five marketplaces contain 286,129 apps, constituting 6.72% of apps in our data set. However, 160,617 (56.1%) out of them are reported by AVTs as PHA, constitutes 15.98% of PHA in our data set. Considering the huge size of these markets, PHA in these marketplaces can affect at least 100 million users. This finding complies with CheckPoint's report, which names China as the country with most mobile threats [180].

PHA susceptibility of different markets. As shown in Table 6.4, we select top 5 PHA families in the selected Android markets. For example, on ANRUAN, among its all

detected PHAs, 29% of them belong to *dowgin* and 2% belong to *wooboo*. Based on the attacks behind its major PHAs, we speculate the susceptible attacks for these markets. To verify our speculation, we upload the recent PHAs of the guessed attacks on each market. For example, all of the markets have the issues of privacy leakage, and there is a considerable amount of Trojan apps in the market WANGYI which put the users under a risk of being remotely controlled.

Enhancement sugguestion for market. Generally, the apps in market undergo upload inspection and online assessment. Google has launched the service BOUNCER to automate the scanning Android apps in GOOGLEPLAY [140], and then turned suspicious apps to human experts for manual check [204]. SLIDEME also provides manual check mechanism for its hosted apps [1]. To reduce the costs of human inspection, we suggest the market to assess the user-uploaded apps with AV tools listed in Table 6.5. For example, for the PHAs contained in GOOGLEPLAY (9.5% of its all apps), 43.5% of these PHAs can be detected by ESET-NOD32, and 31.1% of them can be detected by NANO-Antivirus. We select the optimal AV set with 7 tools inside, such as ESET-NOD32, NANO-Antivirus, Fortinet, Ikarus, Vipre, and Antiy-AVL, and it can detect over 70% of PHA existing in GOOGLEPLAY. With the information in Table 6.5, markets can significantly improve the detection rate for its contained (previously missed) PHAs at the cost of adding several AV tools.

6.4.4 Answer to OQ1 — Authorship of PHA

PHA authors always attempt to produce more PHA samples or variants [205]. The study of PHA authorship facilitates the method of PHA detection based on authors' reputation [205, 206], and identify the authors' behavioral patterns based on the distribution of PHA [207].

Based on the clustering dimension of authorship, we identify 232,536 distinct authors who create 1,004,550 PHA samples in total.

Analysis procedure. All Android apps are required to be signed before they can be installed [208]. Therefore, apps in markets have their own certificate in general. Developers have to provide several items of information to create a public-key certificate,

FIGURE 6.8: Tag cloud for the names of PHA authors

e.g., *private key*, *common name*, *organization* and so on. Hence, the released app can be verified with its public key which is enclosed in the apk file. We extract the information of certificate for all apps, including the public key, the detailed information of issuer, and the detailed information of subject. In general, common name in subject is the name of the owner of the certificate. We extract the common name of the subject as the name of PHA authors, and sketch a tag cloud as shown in Fig. 6.8. The font size of words in tag cloud indicates its frequency, and the distances of words indicate their similarity. PHA authors often use the default one, i.e., *cn* for the acronyms of common name. The two other noticeable names are *Unknown* and *Android Debug*.

Based on the analysis, we conclude several behavioral patterns of PHA authors as follows: We observe that 41,413 PHA samples have the following information of certificate — "/C=US/ST=California/L=Mountain View/O=Android/OU=Android/CN=Android/emailAddress=android@android.com". The certificate, in fact, is a test one provided by AOSP. Generally, the apps with same certificate are regarded to be written by the same author, and can share data and other resources in between. In addition, the app with the same certificate and package name with one existing app can replace the old one. Therefore, it likely leads to a vicious replacement and attacks.

Summary for OQ1.1: Who turns a benign app into PHA? We investigate the top 100 PHA authors, who contribute 29.4% of PHA in total. It reveals that around 84.4% PHA of one single author will be uploaded into two or three Android markets. Moreover, the markets in between present a strong regional relevance, i.e., linguistic or geographic relevance. For example, the author named "Andrew Vasiliu" has uploaded totally 5317

samples onto several markets in *English*, including 4480 (84.3%) into GOOGLEPLAY and 505 (9.5%) into GETJAR. The author named "dns.com.cn Inc." has uploaded totally 3297 samples on several markets in *Chinese*, including 2,544 (77.2%) into ANZHI and 448 (13.6%) into QQ. To sum up, according to our observations from Table 6.6, most authors (96.2%) only create one to two PHAs, and they might manually piggyback benign apps.

Summary for OQ1.2: manners of auto-code generation. In Table 6.6, 3.8% of PHA authors writes 60.9% of PHA in the dataset. According to our inspection to the authors who have produced more than 1,000 PHA samples, we find that many of them are generated in a batched manner:

1. The common way (12.2% of PHAs) is to repackage apps with malicious code [63]. By automatically repackaging benign apps with malicious code, attachers can rapidly disseminate Android PHA in a large scale.

2. App packers are widely used to disseminate PHA. There are products that can encrypt classes.dex in Android apps and evade the detection of AV tools. For example, "secapk" is an encryption tool which is designed against reverse engineering. However, it is widely used in PHA to hide the malicious code. Malicious code is enclosed by a shell which seems harmless but is able to revoke the malicious code [209].

3. There are many frameworks that provide a rapid way to generate apps. For example, we find that one PHA author creates 1,895 apps with the package name matching `^\"com\.cutt\.zhiyue\.android\.app\d+"$` while the content of apps are totally different. These frameworks provide multiple funcationlities, such as push service and geography tracking, and users can only define their basic content of apps. However, we find that some frameworks have collected users' sensitive information on device, and thereby can be employed by cybercriminals to make profits.

TABLE 6.6: The distribution of authorship of Android PHA. The first column refers to the authors which produce PHA in a range of number. For example, "1-10" denote the number of the authors' PHA we can find in the dataset is in the range from 1 to 10; the second column refers to the percentage of the authors over all PHA authors, and; the thrid column refers to the percentage of the PHA written by these authors over all PHA samples.

# Range	**Author**		**Total PHAs**	
	#	%	#	%
1-10	223,702	(96.2%)	374,296	(39.1%)
11-100	8,116	(3.5%)	211,169	(22.0%)
101-500	600	(0.3%)	114,141	(11.9%)
501-1000	52	(–%)	35,208	(3.7%)
>1000	65	(–%)	222,874	(23.3%)

Currently, auto-generation of Android PHA is not well-discussed in existing security reports. Meng [1, 210] *et al.* proposed to apply modularization and evolutionary algorithms to auto-generate variants. However, we have not observed the cases that Meng *et al.* proposed. As the author creates 1,895 apps with different content of apps, the existing auto-generation is more on automation of piggyback process.

6.4.5 Answer to OQ2: Correlation between PHA and vulnerabilities

In this section, we study the correlation between PHA and the security-related events, including vulnerability disclosure and system upgrade, and conclude the reaction sensitivity of Android PHA.

We consider all 1,004,550 PHA in the dataset. In addition, we cluster the PHA on the dimension of PHA family to identify the differences for different families.

Analysis procedure. As both attackers and defenders will react to the security-related events, the number of PHA is inevitably affected. In our analysis, we sketch a time line, in which the x-axis denotes the time and the y-axis denotes the log value on the number

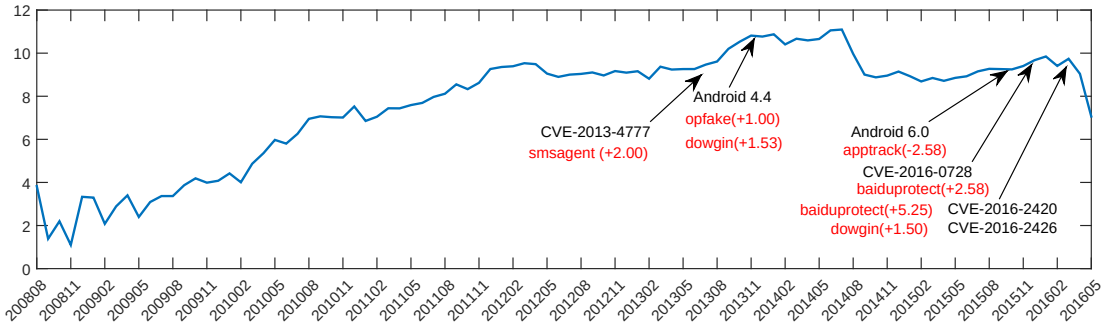

FIGURE 6.9: The chronicle of events and the curve for the number of PHA along with time. The y-axis denotes the logarithmic value of the number of all PHA. The arrows point to time points when events occur, and the affected PHA families.

TABLE 6.7: The influence to PHA of remarkable events in Android. There are two types of events in the experiments, the release of new Android system and the disclosure of vulnerabilities. The column "Event" denotes the influential events, the column "Date" denotes the occurrence date for event, the column "Influence" denotes the caused influence by the event. The value is represented as the affected family and the slope for the change, and the column "Reason" denotes the reason of association.

No	Event	Date	Influence	Reason
1	CVE-2013-4777	2013-07-08	smsagent (+2.00)	smsagent exploits the vul to place a Trojan
2	Android 4.4 (KitKat)	2013-10-31	opfake (+1.0)	opfake depends on vuls of previous versions [211]
3	Android 4.4 (KitKat)	2013-10-31	dowgin (+1.53)	the no. of dowgin reaches its peek in Android 4.4
4	Android 6.0 (Marshmallow)	2015-10-05	apptrack (-2.58)	Runtime perm causes apptrack to be conspicuous
5	CVE-2016-0728, etc.	2015-12-16	baiduprotect (+2.58)	baiduprotect replies on vuls to gain root privilege
6	CVE-2016-2420, etc.	2016-02-18	baiduprotect (+5.25)	baiduprotect replies on vuls to gain root privilege
7	CVE-2016-2426	2016-02-18	dowgin (+1.50)	dowgin may exploit the vul to steal ACCOUNT

of PHA. In order to associate the quantitative change of PHA with the events, we compute the average slope of the quantitative change for the range after the occurrence of a specific event. Let $y = f(x)$ be a curve of the number of PHA along with the time. Assume the occurrence time for the event e is x_0 and the ending point under consideration is $x_1 = x_0 + \triangle x$, we use "average slop" to measure the association between this event e and the number of PHA. The slope at a point x_i can be denoted with the derivative $f_s(x_i)$, and the average slope can be computed as:

$$f_{avg} = \frac{1}{x_1 - x_0} \cap_{x_0}^{x_1} f_s(t)\,dt = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

In this experiment, we set $\triangle x$ to 3 months, and if the average slope $|f_{avg}| \geqslant 1.0$, it is inferred that the event has a significant effect on the number of Android PHA. Fig. 6.9 shows the remarkable events that can cause the "sudden" increase or decrease for PHA.

Summary for OQ2: correlation between PHA and vulnerabilities.

According to the above analysis, we obtain 64 correlations between the number of PHA and events in Android. As shown in Table 6.7, seven highlights refer to the time points of the seven important events. Five PHA families are influenced by these events. PHA *smsagent*, *opfake*, *baiduprotect*, and *dowgin* may all exploit exposed vulnerabilities to carry on special operations. PHA *apptrack* is a tracking app to record users' information from time to time (Interested reader can refer to [198] to view more correlations in between). We obtain the following insights:

- The majority of events (58/64 = 90.6%) that produce correlation are the disclosure of Android vulnerabilities. The rationale is that new PHA outbreaks since the disclosure of exploitable vulnerability, which can request the minimal permissions (i.e., root privilege) but capture the maximal attack capabilities (anything for attack).

- It is commonly acknowledged that some security products may elevate their privilege by exploiting vulnerabilities. For example, *baiduprotect*, which is actually a security software, is classified by other antivirus software. It is because these security software use native code to root the device to obtain more powerful privileges for security protection.

- The correlation analysis helps to quickly identify the associated events at the time point, but the true reasons behind the quantitative change of PHA require the detailed inspection from security analysts. It is rationale to observe that the number of PHAs of privilege escalation has a significant relation with vulnerability events. For example, since the disclosure of CVE-2011-3918, the number of PHA *fakeinst* has increased by 1,170 in three months. However, PHA *fakeinst* has nothing with that vulnerability. It is a kind of PHA that sends SMS to premium-rate number of services and originates back to Feb 24, 2011. Due to its considerable profits, the number has increased in a rapid speed.

6.5 Discussion

Threats to validity. The threats to validity mainly come from three aspects: 1) the representation model for Android apps is a high-level abstraction. It cannot present the detailed behaviors inside, for example the control flow and the data flow of program. However, it can be extracted very effectively, with around 10s to build a representation model for one app. In addition, it has been proved to be qualified in PHA detection [20, 21, 53]. All of these studies extract similar features as representation model, and achieve at least 90% accuracy in PHA detection; 2) the labels given by VIRUSTOTAL are confusing and not 100 percent correct. In particular, the integrated antivirus software may give totally different results for a certain app. Therefore, it can interfere our analysis to some extent. Fortunately, we employ AVCLASS to infer the most likely label for a certain app based on all suggestions of antivirus software, which can reduce the number of wrong labels; 3) the biased data set for Android apps. Apart from GOOGLEPLAY, ANDROZOO and the PHA data sets, the apps in Chinese markets constitute 80.0% of the whole data set. Hence, many of observations and conclusions are suited to the status quo of Chinese markets and customers.

Benefits for security community. From four types of analysis in Section 6.2.2, we gain many useful and insightful conclusions, which can benefit security community. The conclusions from app analysis unveil the corruption of apps during upgrade or evolution, and they can benefit the future detection research; the conclusions from PHA analysis summarize the evolution of characteristics of PHA, correlations between different PHA families, and the propagation of PHA, and so on. They are useful to better understand the current status of Android PHA; the conclusions from authorship analysis identify behavioral patterns of PHA authors, and can facilitate the comprehension of attack techniques and attack targets; the conclusions from market analysis assess the security risk for different markets, which is a good reference for Android developers and users, and a reminder for the owners of market to improve the security inspection.

6.6 Related Work

Android Apps Analysis in Large Scale. There are several studies on large-scale Android app analysis, each of which focuses on a certain problem. Chia *et al.* investigate the usage or abuse of permissions in Android apps. According to the investigation on three popular platforms, they find that a large amount of apps request more permissions than expected. Afonso *et al.* conduct an analysis on native code with dynamic analysis [184]. From over 44k apps which leverage native code, they study the code from multiple aspects such as the behaviors in native code, and call relationship with Java code. ANDRUBIS [182] is proposed to analyze Android malware with multiple techniques including static analysis and dynamic analysis. Based on the obtained features from Android malware, it obtains the prevalence of malicious behaviors in current malware with statistics. CHABADA [212], AUTOREB [213] are designed to investigate the violations of the description of apps against the behaviors inside with natural language processing. Andow *et al.* conduct a study of grayware on Google Play [214], and identify seven categories of grayware as well as existing problems in these grayware. In addition, Allix *et al.* study the behavior patterns of malware authors (e.g., how to write and detect malware) and find several insightful conclusions [215].

Conceptual Model of Android Apps. Conceptual models (e.g., behavior model [44], feature model [1]) can be used to better understand the characteristics of Android apps. Previous efforts have been made to build conceptual models for multiple uses such as risk assessment, malware detection, and behavior prediction. DREBIN [20] depicts Android apps with six types of information, for example, requested permissions and API calls. DROIDSIFT [44] builds a weighted contextual API dependency graph for Android apps. This graph illustrates the dependency relationship between different APIs in Android, which represent the structure of malicious behaviors. APPOSCOPY [42] and MYSTIQUE [1, 210] decouple the components composing malicious behaviors, and build a feature model to depict the behaviors of privacy leakage.

Analysis Report from Security Companies. Most security companies publish annual (or quarterly) security reports to summarize the status quo of the mobile world, and forecast the trend of attacks in future [3, 14, 183, 216]. Generally, these reports provide

a macroscopic view for attacks and malware for the ease of understanding to the public. They provide the statistics of attacks, the proportion of malware, a brief analysis report of a certain malware family, the prevalence of malware and so on. However, our work focuses on the understanding the differences and evolution of the characteristics of Android malware. We aim at finding insightful conclusions from the evolved malware and their correlations, which can directly benefit the detection and comprehension of malware [198]. In addition, since our data set contains over one million malware, which can rival the data set of security companies, the analysis results and findings are convincing and comprehensive.

6.7 Conclusion

We propose two-order representation model to depict the characteristics of Android apps, by combining six dimensions to cluster apps based on interests. Based on the analysis results, we also leverage multiple techniques such as machine learning and statistical inference to identify security-related insights and conclusions from apps. To the best of our knowledge, we have the largest analysis subject in academia to date—4,267,178 apps and 1,004,550 PHA variants. We conduct five types of analysis on this large scale data set. The conclusions and insights obtained from our study should shed light on many areas such as PHA comprehension and detection.

We believe this work is just a starting point for applying data analysis techniques to understand Android security problems. For the future work, we are interested to improve the analysis with fine-grained attack behaviors, complicated evasion techniques, vulnerability exploitation and so on.

7

Conclusions and Future Research

In this chapter, we summarize the research work that we have conducted in the thesis, discuss our future research direction and work.

7.1 Summary of completed work

Along with the development of Android system and its applications, people are facing a critical threat from Android malware. To protect Android from malware, we first investigate state-of-the-art security systems on Android, and the security mechanisms employed in many security systems. We identify many weaknesses and challenges in designing these security systems. To address these weaknesses and challenges, in this thesis, we perform a series of works to understand malware, detection malware, evaluate anti-malware tools, and study the status quo of Android malware in large scale.

First, we propose a framework, named SMART, to learn the semantic model from existing Android malware, and use a combined approach of machine learning and DSA inclusion to detect and classify malware. We perform a differencing analysis on known Android malware, identify the differences and commonalities among them, and then build a semantic model for them. The model facilitates the comprehension of Android malware, and enhances the detection based on refined malware feature. The experiment shows its efficiency and accuracy in malware detection.

Second, we construct a meta-model of which Android malware comprise, and present a framework MYSTIQUE to automatically generate Android malware. The generated malware contains different attack features and evasion features, and thereby is used to evaluate the existing anti-malware tools. In addition, we propose three criteria to evaluate the generated malware: aggressiveness, evasiveness, and detectability. With a multi-objective evolutionary algorithm, we mutate the features of malware and generate optimized malware. We select four kinds of AMTs—static analysis based, dynamic analysis based, machine learning based, and commercial antivirus software to detect generated malware. The result can show the weakness and limitations of these anti-malware tools, and facilitate their improvement in future.

Third, we extend the work MYSTIQUE to MYSTIQUE-S which employ dynamic code loading to dynamically distribute malicious code to the victims. MYSTIQUE-S adopts a client-server architecture. The client app doest not contain any concrete malicious code except receiving the code from the server and dynamically execute it. The server is driven by MYSTIQUE to generate malicious code in terms of devices and attack targets. This kind of attack evades the majority of AMTs and introduces a big risk to Android users in our experiment. In order to mitigate it, we propose several measures to detect these attacks which is a good guide for the improvement of AMTs.

Last, to understand the status quo of Android malware, we conduct a large-scale analysis on 4 million Android apps, and 23.5% of them are malware. We propose a two-order feature model to depict Android malware, and leverage multiple techniques such as machine learning, statistic inference to identify security-related insights and conclusions from apps. To the best of our knowledge, we have the largest analysis subject in

academia to date— 4,267,178 apps and 1,004,550 malware variants. We conduct five types of analysis on this large scale data set (i.e., app analysis, malware family analysis, authorship analysis, market analysis, and vulnerability analysis). The conclusions and insights obtained from our study should shed light on many areas such as malware comprehension and detection.

7.2 Future work

In this thesis, we have an overall understanding of Android malware, and get familiar with multiple techniques to detect malware. However, due to the immensity of Android malware, there are still many security issues that are not covered in this thesis. In future, we are going to conduct the following works:

- **Behavioral pattern of malware authors and malware propagation model.** It is non-trivial to understand behavioral patterns of malware authors. The behavioral patterns of malware authors contain how the authors write, evolve and spread malware. The study can facilitate the detection of malware, and the propagation model can give a beforehand warning for Android markets to prevent attacks in time. In future, we would like to explore behavioral patterns of malware authors from our dataset which contains over 4 million Android apps and 1 million Android malware. According to our preliminary investigation, there are 232,536 distinct malware authors in our dataset. They have created multi-version malware variants and distributed them into different Android markets. This study can unveil the risk and security of Android from the perspective of malware authors.

- **Security flaws in financial apps.** Financial apps are becoming the biggest attack target these years, and the compromise of financial apps can cause a huge lose for individual users or companies. Therefore, a sound and efficient security verification becomes a necessity for current financial apps. We are going to explore the logic inside the financial apps and find out the potential vulnerabilities. In this work, we will propose an attack-defense scenario model to illustrate the threats

and possible mitigation that financial apps are facing. The model can be employed to guide the detection of vulnerabilities and assess the security of these financial apps. We will use static analysis to quickly locate the possible vulnerabilities and employ penetration testing to confirm and replay them.

- **Cross-platform mobile apps analysis.** Android apps are written mostly in Java, which is easier to be decompiled than objective-C in iOS platform. Therefore, It is much easier to analyze apps and detect malware in Android platform. We attempt to conduct a work to study the connection between Android and iOS platform, and use the knowledge in Android platform, to identify attacks and malware existing in iOS platform. In this work, we first build representative features from Android malware, and employ transfer learning to address the similar task for iOS apps.

- **Fuzzing Android system to discover vulnerabilities**. Due to its practicability and usability, fuzzy testing is widely used to detect software vulnerabilities. We attempt to use fuzzy testing [217] to explore vulnerabilities in Android system and its applications. We instrument the whole Android system, and identify all possible inputs of system. We mutate the inputs to generate optimized test cases for the system according to the code coverage of testing, and identify potential implementation or logical vulnerabilities.

- **Big code analysis of Android apps**. Besides the work in Chapter 6, there are many possibilities to learn knowledge from big code. For example, big code is used to learn programming patterns to do de-obfuscation and automate code completion; learning security tactics for development from big code; identifying code smell in Android apps and possible flaws in Android apps, and; energy profiling and hardware usage patterns [218, 219] of Android malware.

Bibliography

[1] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tianming Chen. Mystique: Evolving Android Malware for Auditing Anti-Malware Tools. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 365–376, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4233-9.

[2] IDC. Android and iOS Squeeze the Competition, Swelling to 96.3% of the Smartphone Operating System Market for Both 4Q14 and CY14, According to IDC . `http://www.idc.com/getdoc.jsp?containerId=prUS25450615`, 2015. [Online; accessed 28-Feb-2015].

[3] Symantec. Internet Security Threat Report. Technical report, Apr 2016.

[4] Guozhu Meng, Yang Liu, Jie Zhang, Alexander Pokluda, and Raouf Boutaba. Collaborative Security: A Survey and Taxonomy. *ACM Computing Surveys*, 48 (1):1:1–1:42, July 2015. ISSN 0360-0300.

[5] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0.

[6] Smartphone OS Market Share, 2016 Q2. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, 2016.

[7] Carlos A. Castillo. Android Malware Past, Present, and Future. Technical report, 2012.

[8] TrendMicro Inc. A Brief History of Mobile Malware. Technical report, 2012.

[9] Myrto Arapinis, Loretta Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New Privacy Issues in Mobile Telephony: Fix and Verification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 205–216, 2012.

[10] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[11] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, pages 95–109, Washington, DC, USA, 2012.

[12] Xuxian Jiang and Yajin Zhou. *Android Malware*. Springer, June 2013.

[13] Yajin Zhou and Xuxian Jiang. An Analysis of the AnserverBot Trojan. Technical report, Sep 2011.

[14] McAfee. McAfee Labs Threats Report. Technical report, Mar 2016.

[15] Joji Hamada. Simplocker: First Confirmed Ransomware for Android. `http://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android`.

[16] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *17th European Symposium on Research in Computer Security (ESORICS)*, volume 7459, pages 37–54, 2012.

[17] Jian Chen, Manar H. Alalfi, Thomas R. Dean, and Ying Zou. Detecting android malware using clone detection. *J. Comput. Sci. Technol.*, 30(5):942–956, 2015.

[18] DARELL J. J. TAN SUFATRIO, TONG-WEI CHUA, and VRIZLYNN L. L. THING. Securing Android: A Survey, Taxonomy, and Challenges. *ACM Computing Surveys*, 47(4), May 2015.

[19] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Computing Surveys (CSUR)*, 49(3):55:1–55:30, October 2016.

[20] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *The Network and Distributed System Security Symposium (NDSS)*, 2014.

[21] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, and Andreas Zeller. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[22] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, 2014.

[23] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5.

[24] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating Malicious and Benign Mobile App Behavior

Under Contexts. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2014.

[25] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–6, Berkeley, CA, USA, 2010.

[26] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 49–54, 2011.

[27] Omer Tripp and Julia Rubin. A Bayesian Approach to Privacy Enforcement in Smartphones. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 175–190, 2014.

[28] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable Race Detection for Android Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 332–348, 2015.

[29] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. Generating Test Cases to Expose Concurrency Bugs in Android Applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–653, 2016.

[30] Guang Gong. Fuzzing Android System Services by Binder Call to Escalate Privilege. In *BlackHat USA 2015*, 2015.

[31] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia (MoMM)*, pages 68:68–68:74, 2013.

[32] Wontae Choi, George Necula, and Koushik Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 623–640, 2013.

[33] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. GUITAR: Piecing Together Android App GUIs from Memory Images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 120–132, 2015.

[34] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 559–570, 2016.

[35] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*, pages 235–245, 2009.

[36] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX conference on Security*, 2011.

[37] Justin Sahs and Latifur Khan. A Machine Learning Approach to Android Malware Detection. In *EISIC*, pages 141–147, 2012.

[38] Naser Peiravian and Xingquan Zhu. Machine Learning for Android Malware Detection Using Permission and API Calls. In *ICTAI*, pages 300–305, 2013.

[39] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *USENIX Security*, pages 543–558, 2013.

[40] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[41] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-Resilient, Efficient, and Accurate Detection and Family Identification of Android Malware. Technical report, Oct. 2015.

[42] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 576–587, 2014.

[43] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Machine Learning-Based Malware Detection for Android Applications: History Matters! Technical report, May 2014.

[44] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *CCS*, Scottsdale, AZ, November 2014.

[45] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based Malware Detection System for Android. In *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile devices (SPSM)*, pages 15–26, 2011.

[46] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. NetworkProfiler: Towards Automatic Fingerprinting of Android Apps. In *IEEE INFOCOM*, pages 809–817, 2013.

[47] VirusShare. `http://www.virusshare.com`.

[48] Essam Al Daoud and Iqbal H. Jebril and Belal Zaqaibeh. Computer Virus Strategies and Detection Methods. 1(2), 2008.

[49] Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R. Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *The Network and Distributed System Security Symposium (NDSS)*, 2013.

[50] Sanjeev Das, Hao Xiao, Yang Liu, and Wei Zhang. Online Malware Defense Using Attack Behavior Model. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1322–1325, May 2016.

[51] Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware. *IEEE Transactions on Information Forensics and Security (TIFS)*, 11(2):289–302, 2016.

[52] Yinxing Xue, Junjie Wang, Yang Liu, Hao Xiao, Jun Sun, and Mahinthan Chandramohan. Detection and Classification of Malicious JavaScript via Attack Behavior Modelling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 48–59, 2015.

[53] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *SecureComm*, volume 127, pages 86–103, 2013.

[54] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security (AISec)*, pages 45–54, 2013.

[55] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Checking Interation-Based Declassification Policies for Android Using Symbolic Execution. Technical report, 2009.

[56] Kai Chen, Peng Wang, Yeonjoon Lee, Xiaofeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding Unknown Malice in 10 Seconds: Mass

Vetting for New Threats at the Google-Play Scale. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 659–674, 2015.

[57] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *The Network and Distributed System Security Symposium (NDSS)*, 2015.

[58] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. DroidScribe: Classifying Android Malware Based on Runtime Behavior. In *MoST*, 2016.

[59] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. Semantic Modelling of Android Malware for Effective Malware Comprehension, Detection, and Classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 306–317, 2016. ISBN 978-1-4503-4390-9.

[60] VirusTotal - Free Online Virus, Malware and URL Scanner. `https://www.virustotal.com`, 2015.

[61] Marcos Sebastian, Richard Rivera, Platon Kotzias, and Juan Caballero. AVCLASS: A Tool for Massive Malware Labeling. In *RAID*, 2016.

[62] McAfee Labs. Threats Predictions. Technical report, 2015.

[63] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of “Piggybacked" Mobile Applications. In *Proceedings of the second ACM conference on Data and Application Security and Privacy (CODASPY)*, pages 185–196, 2013.

[64] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy (CODASPY)*, pages 317–326, 2012.

[65] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious

Behaviors in Android Applications. In *ESORICS*, volume 8712, pages 163–182. Springer International Publishing, 2014. ISBN 978-3-319-11202-2.

[66] Junjie Wang, Yinxing Xue, Yang Liu, and Tian Huat Tan. JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, pages 109–120, New York, NY, USA, 2015. ISBN 978-1-4503-3245-3.

[67] Annamalai Narayanan, Guozhu Meng, Yang Liu, Jinliang Liu, and Lihui Chen. Contextual Weisfeiler-Lehman Graph Kernel for Malware Detection. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 4701–4708, July 2016.

[68] Annamalai Narayanan, Yang Liu, Lihui Chen, and Jinliang Liu. Adaptive and Scalable Android Malware Detection through Online Learning. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 2484–2491, July 2016.

[69] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. 41(9):866–886, 2015.

[70] Hila Peleg, Sharon Shoham, Eran Yahav, and Hongseok Yang. Symbolic Automata for Static Specification Mining. In *LNCS*, volume 7935, pages 63–83. Springer Berlin Heidelberg, 2013.

[71] Gabel, Mark and Jiang, Lingxiao and Su, Zhendong. Scalable Detection of Semantic Clones. In *ICSE*, pages 321–330, 2008.

[72] AppBrain. Android Library Statistics. `http://www.appbrain.com/stats/libraries`, 2015. [Online; accessed 02-Jan-2015].

[73] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *GbR*, 2001.

[74] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In *2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 71–82, 2015.

[75] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *WISEC*, pages 13–24, New York, NY, USA, 2013.

[76] Domagoj Babić, Daniel Reynaud, and Dawn Song. Malware Analysis with Tree Automata Inference*. In *23rd International Conference on Computer Aided Verification (CAV)*, pages 116–131, 2011.

[77] Mila Dalla Preda, Roberto Giacobazzi, Arun Lakhotia, and Isabella Mastroeni. Abstract Symbolic Automata: Mixed Syntactic/Semantic Similarity Analysis of Executables. In *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 329–341, 2015.

[78] Guillaume Bonfante, Jean-Yves Marion, and Thanh Dinh Ta. Malware Message Classification by Dynamic Analysis. In *International Symposium on Foundations and Practice of Security (FPS)*, pages 112–128, 2015.

[79] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 62–81, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37299-5. doi: 10.1007/978-3-642-37300-8_4.

[80] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *36th International Conference on Software Engineering (ICSE)*, pages 175–186, New York, NY, USA, 2014.

[81] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps

with Permission Use Analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS)*, pages 611–622, 2013.

[82] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Merwe, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. Towards Model Checking Android Applications. *IEEE Transactions on Software Engineering (TSE)*, PP(99), 2017.

[83] Xiaoning Du, Yang Liu, and Alwen Tiu. Trace-Length Independent Runtime Monitoring of Quantitative Policies in LTL. In Nikolaj Bjørner and Frank de Boer, editors, *20th International Symposium on Formal Methods (FM)*, pages 231–247, Cham, 2015. Springer International Publishing.

[84] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of Interface Specifications for Java Classes. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 98–109, 2005.

[85] Hao Xiao, Jun Sun, Yang Liu, Shang-Wei Lin, and Chengnian Sun. TzuYu: Learning Stateful Typestates. In *Automated Software Engineering (ASE)*, pages 432–442, 2013.

[86] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 997–1016, 2012.

[87] Xuxian Jiang. Security Alert: New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets. `http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html`, 2011. [Online; accessed 13-Dec-2014].

[88] Kai Chen, Peng Wang, Yeonjoon Lee, Xiaofeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *24th USENIX Conference on Security Symposium*, pages 659–674, aug. 2015.

[89] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *CASCON*, pages 13–, 1999.

[90] Iman Keivanloo, Feng Zhang, and Ying Zou. Threshold-free Code Clone Detection for a Large-scale Heterogeneous Java Repository. In *SANER*, pages 201–210, 2015.

[91] Da-Fei Feng and RussellF. Doolittle. Progressive Sequence Alignment as a Prerequisitetto Correct Phylogenetic Trees. *Journal of Molecular Evolution*, 25(4): 351–360, 1987. ISSN 0022-2844.

[92] Yun Lin, Zhenchang Xing, Yinxing Xue, Yang Liu, Xin Peng, Jun Sun, and Wenyun Zhao. Detecting Differences Across Multiple Instances of Code Clones. In *ICSE*, pages 164–174, 2014.

[93] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.

[94] Frank Harary and RobertZ. Norman. Some Properties of Line Digraphs. *Rendiconti del Circolo Matematico di Palermo*, 9(2):161–168, 1960. ISSN 0009-725X. doi: 10.1007/BF02854581. URL `http://dx.doi.org/10.1007/BF02854581`.

[95] Eric Lafortune. ProGuard. `http://proguard.sourceforge.net/`, 2015.

[96] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and Data-flow Analysis of While-programs. *TOPLAS*, 7(1):37–61, January 1985.

[97] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 217–228, 2012.

[98] Curse of dimensionality. http://en.wikipedia.org/wiki/Curse_of_dimensionality, 2015.

[99] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[100] M. De Wulf, L. Doyen, T.A. Henzinger, and J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Computer Aided Verfication (CAV)*, volume 4144, pages 17–30, 2006.

[101] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Exploitation and threat analysis of open mobile devices. *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems - ANCS '09*, page 20, 2009. doi: 10.1145/1882486.1882493.

[102] Yang Liu, Jun Sun, and Jin Song Dong. Developing Model Checkers Using PAT. In *8th Internatoinal Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 371–377, 2010.

[103] Ting Wang, Songzheng Song, Jun Sun, Yang Liu, JinSong Dong, Xinyu Wang, and Shanping Li. More Anti-chain Based Refinement Checking. In *14th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM)*, volume 7635, pages 364–380, 2012.

[104] Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shaojie Zhang, and Jinsong Dong. Verifying Linearizability via Optimized Refinement Checking. *IEEE Transactions on Software Engineering*, 39(7):1018–1039, 2013.

[105] Ting Wang, Jun Sun, Xinyu Wang, Yang Liu, Yuanjie Si, Jin Song Dong, Xiaohu Yang, and Xiaohong Li. A Systematic Study on Explicit-State Non-Zenoness Checking for Timed Automata. *IEEE Transactions on Software Engineering (TSE)*, 41(1):3–18, Jan 2015.

[106] SMART: Semantic Modelling of Android Attacks. https://sites.google.com/site/droidsmat2015/, 2015.

[107] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.

[108] Saurabh Oberoi, Weilong Song, and Amr M. Youssef. AndroSAT: Security Analysis Tool for Android Applications. In *SecureWare*, 2014.

[109] AndroidDrawer.com - APK Download of Free Android Apps. `http://www.androiddrawer.com/`, 2014.

[110] AnZhi. `http://www.anzhi.com/`, 2014.

[111] APKMirror - #APKPLZ #SOONBACKANSWER. `http://www.apkmirror.com/`, 2015.

[112] Android Apps, Download APK, Android Applications, Android APK. `http://www.appsapk.com/`, 2014.

[113] ChinaMobile. `http://mm.10086.cn/`, 2015.

[114] Coolapk. `http://www.coolapk.com/`, 2015.

[115] EOEMarket. `http://www.eoemarket.com/`, 2015.

[116] F-Droid: Free and Open Source Android App Repository. `http://f-droid.org`, 2014.

[117] Flyme. `http://app.flyme.cn/`, 2015.

[118] GetJar - Download Free Apps, Games and Themes APK. `http://www.getjar.com/`, 2015.

[119] GFan. `http://apk.gfan.com/`, 2015.

[120] Google Play. `http://play.google.com/store/`, 2014.

[121] Huawei. `http://appstore.huawei.com/`, 2015.

[122] SlideME | Android Apps Market: Download Free & Paid Android Applications. `http://slideme.org/`, 2014.

[123] Wandoujia. http://www.wandoujia.com/apps/, 2015.

[124] Netease. http://m.163.com/android/, 2015.

[125] Mi. http://app.mi.com/, 2015.

[126] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *AsiaCCS*, pages 329–334, 2013.

[127] 10 Years of Mobile Malware Whitepaper. http://www.fortinet.com/sites/default/files/whitepapers/10-Years-of-Mobile-Malware-Whitepaper.pdf, 2014.

[128] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS*, 2011.

[129] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2014.

[130] Hendra Gunadi and Alwen Tiu. Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System. *CoRR*, abs/1311.2362, 2013.

[131] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Nov 1990.

[132] David A. Mundie and David M. McIntire. An Ontology for Malware Analysis. In *ARES*, pages 556–558, 2013.

[133] Mystique | Evolving Android Malware for Auditing Anti-Malware Tools. https://sites.google.com/site/malwareevolution/.

[134] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. On the value of user preferences in search-based software engineering: a case study in software product

lines. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 492–501, 2013.

[135] Tian Huat Tan, Yinxing Xue, Manman Chen, Jun Sun, Yang Liu, and Jin Song Dong. Optimizing selection of competing features via feedback-directed evolutionary algorithms. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 246–256, 2015.

[136] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and Xiaoyang Sean Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS)*, pages 1043–1054, 2013.

[137] Activity | Android Developer. `http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle`.

[138] Hisao Ishibuchi, Noritaka Tsukamoto, and Yusuke Nojima. Evolutionary Many-Objective Optimization: A Short Review. In *CEC*, pages 2419–2426, 2008.

[139] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S. Foster, and Michael R. Clarkson. Checking Interation-Based Declassification Policies for Android Using Symbolic Execution. Technical Report arXiv:1504.03711v2, 2015.

[140] Hiroshi Lockheimer. Android and Security: Official Google Mobile Blog. `http://googlemobile.blogspot.sg/2012/02/android-and-security.html`, 2012.

[141] Eunice Kim. Creating Better User Experiences on Google Play. `http://android-developers.blogspot.ro/2015/03/creating-better-user-experiences-on.html`, 2015.

[142] TorrApk - Alternative Android App Store for Free Applications. `https://www.torrapk.com/en`.

[143] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *The Network and Distributed System Security Symposium (NDSS)*, 2014.

[144] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative TCP Sequence Number Inference Attack: How to Crack Sequence Number Under a Second. In *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*, pages 593–604, 2012.

[145] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security*, pages 1037–1052, 2014.

[146] YoungHan Choi, TaeGhyoon Kim, SeokJin Choi, and CheolWon Lee. Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis. In *FGIT*, pages 160–172, 2009.

[147] Emre Aydogan and Sevil Sen. Automatic Generation of Mobile Malwares Using Genetic Programming. In *Applications of Evolutionary Computation*, volume 9028, 2015.

[148] Andrea Cani, Marco Gaudesi, Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. Towards Automated Malware Creation: Code Generation and Code Integration. In *SAC*, pages 157–160, 2014.

[149] Mihai Christodorescu and Somesh Jha. Testing Malware Detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44, 2004.

[150] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *EuroSec*, pages 5:1–5:6, 2014.

[151] Dominik Maier, Tilo Müller, and Mykola Protsenko. Divide-and-Conquer: Why Android Malware cannot be stopped. In *ARES*.

[152] Federico Maggi, Andrea Valdi, and Stefano Zanero. AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors. In *Proceedings of the Third ACM workshop on Security and Privacy in Smartphones & Mobile devices (SPSM)*, pages 49–54, 2013.

[153] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. In *Proceedings of the 9th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 82–101, 2013.

[154] Heqing Huang, Kai Chen, Chuangang Ren, Peng Liu, Sencun Zhu, and Dinghao Wu. Towards Discovering and Understanding Unexpected Hazards in Tailoring Antivirus Software for Android. In *AsiaCCS*, pages 7–18, 2015.

[155] AV-TEST. `https://www.av-test.org/`.

[156] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.

[157] Kyo Chul Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4), 2002.

[158] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS'14*, 2014.

[159] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring Dynamic Software Product Lines. In *GPCE*, pages 3–12, 2011.

[160] Don S. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, pages 7–20, 2005.

[161] Mark Harman, Yue Jia, Jens Krinke, William B. Langdon, Justyna Petke, and Y. Zhang. Search based software engineering for software product line engineering: a survey and directions for future work. In *SPLC*, pages 5–18, 2014.

[162] Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. A Taxonomy of Privilege Escalation Attacks in Android Applications. *Int. J. Secur. Netw.*, 9(1): 40–55, February 2014.

[163] Mystique-S | Automated Android Malware Generation by Evolution and Dynamic Loading Technique. https://sites.google.com/site/malwareasaservice/.

[164] Adam Turner. Malware hijacks big four australian banks' apps, steals two-factor sms codes. https://t.co/ud5P7C8Zzq, 2016.

[165] ECMA International. ECMAScript 2015 Language Specification. Technical report, 2015.

[166] Pim van den Broek. Optimization of product instantiation using integer programming. In *SPLC*, pages 107–112, 2010.

[167] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[168] Xposed Module Repository. http://repo.xposed.info/.

[169] Cplex. http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud, 2016.

[170] ESET. The Rise of Android Ransomware. Technical report, 2014.

[171] TechEye. Android malware MisoSMS one of the largest botnets to date. http://www.tgdaily.com/security-brief/83076-android-malware-misosms-one-of-the-largest-botnets-to-dat

[172] Ali Zand, Giovanni Vigna, Xifeng Yan, and Christopher Kruegel. Extracting probable command and control signatures for detecting botnets. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC)*, pages 1657–1662, 2014.

[173] Sebastián García, Alejandro Zunino, and Marcelo Campo. Survey on network-based botnet detection methods. *Security and Communication Networks*, 7(5): 878–903, 2014.

[174] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 37–48, 2015.

[175] Rafael Fedler, Marcel Kulicke, and Julian Schütte. An Antivirus API for Android Malware Recognition. In *MALWARE*, pages 77–84, 2013.

[176] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth Attacks: An Extended Insight into the Obfuscation Effects on Android Malware. *Computers & Security*, 51, 2015.

[177] Symantec. Internet Security Threat Report. Technical report, Apr 2014.

[178] Google Inc. Android Security 2015 Year In Review. Technical report, Apr 2016.

[179] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *MSR*, pages 468–471, 2016.

[180] Check Point Software Technoligies LTD. Check Point - 2016 Security Report. Technical report.

[181] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. AndRadar: Fast Discovery of Android Applications in Alternative Markets. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 51–71, 2014.

[182] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis – 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *BADGERS*, pages 3–17, 2014.

[183] F-Secure. Threat Report 2015. Technical report, Dec 2015.

[184] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[185] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. Is this App Safe? A Large Scale Study on Application Permission and Risk Signals. In *WWW*, pages 311–320, 2012.

[186] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, Xiaofeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *IEEE Symposium on Security and Privacy (S&P)*, number 20, pages 357–376, 2016.

[187] Vaibhav Rastogi, Rui Shao, Yan Chen Xiang Pan, Shihong Zou, and Ryan Riley. Are these Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Inferfaces. In *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[188] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Conference on Security (SEC)*, pages 527–542, 2013. ISBN 978-1-931971-03-4.

[189] Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. RiskMon: Continuous and Automated Risk Assessment of Mobile Applications. In *Proceedings of the 4th ACM conference on Data and application security and privacy (CODASPY)*, pages 99–110, 2014.

[190] Wei Wang, Xing Wang, Dawei Feng, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. Exploring Permission-Induced Risk in Android Applications for Malicious Application Detection. *IEEE Transactions on Information Forensics and Security (TIFS)*, 9(11):1869–1882, November 2014.

[191] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 281–294, 2012.

[192] Guang Gong. Fuzzing Android System Services by Binder Call. In *BlackHat*, 2015.

[193] Huan Feng and Kang G. Shin. BinderCracker: Assessing the Robustness of Android System Services. In *arXiv*, 2016.

[194] Trend Micro. A Brief History of Mobile Malware. Technical report, 2012. URL `https://countermeasures.trendmicro.eu/wp-content/uploads/2012/02/History-of-Mobile-Malware.pdf`.

[195] Veo Zhang. 'GODLESS' Mobile Malware Uses Multiple Exploits to Root Devices. `http://blog.trendmicro.com/trendlabs-security-intelligence/godless-mobile-malware-uses-multiple-exploits-root-devices/`, 2016.

[196] Google Inc. Android NDK | Android Developers. `https://developer.android.com/ndk/index.html`, 2008.

[197] Charlie Curtsinger, Benjamin Livshits, Benjain Zorn, and Christian Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Conference on Security*, 2011.

[198] Guozhu Meng and Yinxing Xue. The Vampire Diaries for Android — A Large Scale Android Malware Analysis. `https://sites.google.com/site/largescaleanalysis/`, 2016.

[199] Wikipedia. Cosine Similarity. https://en.wikipedia.org/wiki/Cosine_similarity.

[200] Clemens Orthacker, Peter Teufl, Stefan Kraxberger, Günther Lackner, Michael Gissing, Alexander Marsalek, Johannes Leibetseder, and Oliver Prevenhueber. Android Security Permissions – Can We Trust Them? In *IEEE S&P*, pages 40–51, 2012.

[201] Permission based Android Security: Issues and Countermeasures. *Computers & Security*, 43:205–218, 2014.

[202] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *USENIX Security*, pages 499–514, August 2015.

[203] Xuxian Jiang. An Evaluation of the Application ("App") Verification Service in Android 4.2. https://www.csc.ncsu.edu/faculty/jiang/appverify/, 2012.

[204] Sarah Perez. App Submissions On Google Play Now Reviewed By Staff, Will Include Age-Based Ratings.

[205] OBA2: An Onion approach to Binary code Authorship Attribution. *Digital Investigation*, 11:94–103, 2014.

[206] Ivan Krsul and Eugene H. Spafford. Authorship analysis: Identifying the author of a program. Technical report, Computers and Security, 1996.

[207] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. Who Wrote This Code? Identifying the Authors of Program Binaries. In *ESORICS*, pages 172–189, 2011.

[208] Google Inc. Sign Your App. https://developer.android.com/studio/publish/app-signing.html.

[209] Tim Strazzere and Jon Sawyer. Android Hacker Protection Level 0. Technical report, 2014. URL https://www.defcon.org/images/defcon-22/dc-22-presentations/Strazzere-Sawyer/

DEFCON-22-Strazzere-and-Sawyer-Android-Hacker-Protection-Level-UPDAT
pdf.

[210] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique. *IEEE Transactions on Information Forensics and Security (TIFS)*, 12(7):1529–1544, July 2017.

[211] Duncan Alfreds. Advertising malware targets SA smartphones. http://www.fin24.com/Tech/Cyber-Security/advertising-malware-targets-sa-smartphones-20160323, 2016.

[212] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking App Behavior Against App Descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 1025–1035, New York, NY, USA, 2014. ACM.

[213] Deguang Kong, Lei Cen, and Hongxia Jin. AUTOREB: Automatically Understanding the Review-to-Behavior Fidelity in Android Applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 530–541, 2015.

[214] Benjamin Andow, Adwait Nadkarni, Blake Bassett, William Enck, and Tao Xie. A Study of Grayware on Google Play. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 224–233, May 2016.

[215] Kevin Allix, Quentin Jerome, Tegawendé F. Bissyandé, Jacques Klein, Radu State, and Yves Le Traon. A Forensic Analysis of Android Malware – How is Malware Written and How It Could Be Detected? In *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference (COMPSAC)*, pages 384–393, 2014. ISBN 978-1-4799-3575-8.

[216] ESET. ESET Security Report. Technical report, Apr 2016.

[217] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. In *38th IEEE Symposium on Security and Privacy (S&P 2017)*, 2017.

[218] Liang He, Guozhu Meng, Yu Gu, Cong Liu, Jun Sun, Ting Zhu, Yang Liu, and Kang G. Shin. Battery-Aware Mobile Data Service. *IEEE Transactions on Mobile Computing (TMC)*, PP(99), 2016.

[219] Liang He, Eugene Kim, Kang G. Shin, Guozhu Meng, and Tian He. Battery state-of-health estimation for mobile devices. In *Proceedings of the 8th International Conference on Cyber-Physical Systems (ICCPS)*, pages 51–60, 2017.