ANALYSIS TECHNIQUES FOR INTRA AND INTER APP(S) IN ANDROID

Ph.D. Thesis

SHWETA BHANDARI
ID No. 2014RCP9508

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY, JAIPUR
NOVEMBER 2018

Analysis Techniques for Intra and Inter App(s) in Android

Submitted in
fulfillment of the requirements for the degree of

Doctor of Philosophy

by

Shweta Bhandari
ID: 2014RCP9508

Under the Supervision of
Prof. Manoj Singh Gaur, MNIT Jaipur
Prof. Vijay Laxmi, MNIT Jaipur
Dr. Akka Zemmari, University of Bordeaux, France

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY, JAIPUR
November 2018

Declaration

I, **Shweta Bhandari**, declare that this thesis titled, "**Analysis Techniques for Intra and Inter App(s) in Android**" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Ph.D. degree at Malaviya National Institute of Technology, Jaipur.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Malaviya National Institute of Technology, Jaipur or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this Dissertation is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself, jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Date: Shweta Bhandari

(2014RCP9508)

CERTIFICATE

This is to certify that the thesis entitled "**Analysis Techniques for Intra and Inter App(s) in Android**" being submitted by **Shweta Bhandari (2014RCP9508)** is a bona-fide research work carried out under my supervision and guidance in fulfillment of the requirement for the award of the degree of Doctor of Philosophy in the Department of Computer Science and Engineering, Malaviya National Institute of Technology, Jaipur, India. The matter embodied in this thesis is original and has not been submitted to any other University or Institute for the award of any other degree.

Place: Jaipur
Date:

(Supervisors)

Prof. Manoj Singh Gaur
Professor
Department of Computer Science and Engineering
MNIT Jaipur

Prof. Vijay Laxmi
Professor
Department of Computer Science and Engineering
MNIT Jaipur

Dr. Akka Zemmari
Associate Professor
Bordeaux Laboratory of Research in Computer Science
University of Bordeaux, France

Acknowledgements

Foremost, I would like to express my sincere gratitude to my principal advisor **Prof. Manoj Singh Gaur**, IIT Jammu, India, for his continuous support to my research, for his patience, motivation, enthusiasm. I could not have imagined having a better advisor and mentor for my Ph.D. Secondly, I would like to express my gratitude to my external supervisor **Dr. Akka Zemmari**, University of Bordeaux, France, who helped me through his profound feedback and comments that polished my ideas. His guidance helped me in all the time of research and writing of this thesis.

My special thanks to **Dr. Vijay Laxmi** for teaching me the real meaning of research and always motivated to research the solution until it reaches its full potential. The progress, I achieved in research was mostly due to her thorough and critical reviews of my manuscripts and thesis. She helped me to clarify and organize my research, which was an essential step to preparing for my thesis work.

I would like to express my gratitude to **Dr. Partha S. Roop**, University of Auckland, New Zealand, and **Dr. Frederic Herbreteau**, University of Bordeaux, France, who introduced me to Formal Verification. The progress, I made in this direction is not possible without their help. Thank you, both for reviewing my research papers and providing valuable suggestions. Also, **Dr. Girdhari Singh**, **Dr. Emmanuel S Pilli** and **Dr. Vijay Janyani** deserve special thanks as my research committee members and advisors.

I was fortunate to meet wonderful friends and fellow researchers Rishab Gupta, Lovely Sinha, Rekha, Smita Naval, Wafa Ben Jaballah, Vineeta Jain, Joey Pinto, Jyoti Gajrani, Shweta Saharan, Garima Garg and Pranjal for helping me by their positive and important suggestions during my low phases in both research and personal life.

Last but not the least, I could not have finished my study without the enduring support of my family. I deeply appreciated the love and support of my husband, Dinesh, who supported me in every possible way. Special thanks to my mother, father, brother, my mother-in-law, and father-in-law for their direct and indirect support and love. I am also grateful to my other family members and friends who have supported me along the way.

A very special gratitude goes out to Ministry of Electronics and Information Technology, Government of India for providing the funding for the work and Department of Computer Science and Engineering, MNIT Jaipur for providing all the other resources required for the work.

Place: Jaipur
Date: Shweta Bhandari

Abstract

With almost universal digital convergence, mobile devices provide an attractive attack surface for cyber thieves as the devices hold personal details and have potential capabilities for eavesdropping. Android is the most popular mobile operating system and hence, is the target of malicious hackers who use the Android app as a tool to gain access to private information. The consequences of these attacks lead to damages that could be monetary or nonmonetary (loss of reputation, physical or mental pain or suffering).

Academic researchers and commercial anti-malware companies are working vigorously to detect malicious apps by proposing detection tools. These tools fail when the malicious behaviour is scattered across more than one app. Also, Android framework is not designed to protect the information that is going outside an app. In such a scenario, individual app shall appear benign whereas it may leak private information in the presence of another specific app(s). This phenomenon of data/information leakage is termed as collusion, and involved apps are termed as colluding apps. In this thesis, we design and develop collusion analysis and detection techniques for Android malware. We also propose formal methods based analysis for the detection of maliciousness causing inter-app information leakage.

Firstly, we propose *DRoid Analyst COmbo (DRACO)*, an Android app analysis mechanism to inspire on-device analysis. DRACO extracts permissions, intent-filters, requested hardware, accessed network addresses along with restricted, suspicious and unused API calls. In the end, machine learning is used to classify the app as malicious or benign based on the extracted features. It generates D-Score which is derived from the probability distribution of the *apk* features towards maliciousness. Due to computational restrictions, thorough app analysis is not possible on-device. Therefore, to increase the code coverage and detection accuracy, we have further focused towards off-device analysis techniques.

Next, we propose a *Semantic AWare AndrOid MalwaRe Detector (SWORD)*. It encapsulates the semantics of Android apps using Asymptotic Equipartition Property (AEP) which are further quantified to detect the malicious apps. Users are extending functionalities of their devices by installing apps from various developers and vendor in an open ecosystem. These apps may misuse the sensitive information stored on the phone or obtained from the sensors to violate user's privacy. There is a need to analyze the actual behaviour of apps with regard to privacy.

Static data flow analysis is a means for automatically enumerating the data flow inside a program. We propose *Android App Analysis via Data Flow: FlowMine* that considers the data flow path from a data source to a data sink, where 'source' is a non-constant data that marks the beginning of the path, and 'sink' is the resource where the data reaches. We analyzed the data flow paths in 2800 benign against 15000 malicious apps and assigned weights to each path, which is the absolute difference between its use in benign and malicious samples.

In order to capture information leakage paths scattered across multiple apps, we propose *Intersection Automata based Model for Android Application Collusion.* In this technique, we represent apps communication through 'application automaton' and identify policies based on the violation of permission model through 'policy automaton.' The intersection of the two automata will detect the presence of collusion. The proposed technique has high false positive.

Next, we propose *Large Scale Klepto Apps Analysis: SneakLeak+*. It statically analyzes the reverse engineered intermediate code of each app, extracts relevant security information, and represents the extracted information into a compact form suitable for formal verification. Formal analysis is used to verify the presence/absence of potential inter-app communication-based leakage. To maintain scalability of the proposed method, we build an abstract model of the apps that represent only potential leaks. To demonstrate the efficacy and scalability of our proposal, we conduct a set of experiments on $11,000$ apps from Google Play Store and benchmark datasets. Our experiments show that SneakLeak+ achieves highest precision (100%), highest recall (93.3%) and highest F-measure (0.97) as compared to existing state-of-the-art approaches.

Currently, there is no standard app dataset available to verify efficacy and scalability of methods dealing with collusion detection. Therefore, we developed 64 wide-ranging apps exhibiting collusion as our benchmark dataset, now, available as open-source. We have also formally defined Dangerous Permissions, Sensitive API Calls, Inter-Component Communication Methods and Resource-Permission Map function that is further used to define Communication, Communication Path, Sensitive Communication Path and Application Collusion.

Dedications

*dedicate my PhD Thesis to my mother **Seema**, father **Kamal** who taught me the lessons of discipline, honesty and sincerity and my husband **Dinesh** without whom, I would not have been able to pursue my doctoral research with utmost dedication.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With ever-increasing technological advancements, the smartphone has become one of the essential components of our daily life. It has replaced not only the traditional phones but also the watch, the alarm clock, the calendar, the organizer, the notebook, the camera, and many other things. The popularity of smartphone is mainly due to their extensibility that allows the user to download and install programs, called apps, aimed at a particular function. As a consequence, more and more data is processed on smartphones, including private, sensitive and confidential data. According to a study made by Statista [1], by 2020, around 2.87 billion smartphone users will exist around the globe. Nowadays, Android is the most popular smartphone OS that has captured 85% of the worldwide smartphone market, leaving its competitor iOS, Windows OS and others far behind [2].

With this popularity, it is also becoming the key target for malware adversaries. Android is vulnerable to many security risks. According to the recent OWASP mobile security report [3], out of 91 reported security risks, 85 are recorded to be present in Android. This makes Android security a serious concern. These risks are outcome of either maliciously exploiting the legitimate procedures provided by android such as ICC, or taking advantage of unchecked processes occurring in the system. Following are some of the key reasons:

- Android offers open-sourced ecosystem that offers the information and source code needed to create custom variants of the platform.

- Android mobiles are available at very affordable cost as there is no monopoly, where one industry player could put restrictions.
- Hosting third-party apps are extremely affordable on Google Play Store.

Although, Google started a service, called 'Bouncer' to automatically scan the entire Play Store (and all newly uploaded apps), malware writers keep on finding new ways to circumvent the screening mechanism. According to a report published in September 2017 [4], there are dozens of malicious apps present on Play Store that sent fraudulent premium text messages and charged people for fake services. Google also added 'app verification,' a security feature in Android OS, yet banking trojans and other malicious Android apps are still spreading. Recent years have witnessed, an upsurge of malicious programs in the form of Android apps. For instance, Android malware like *Cloak and Dagger* attack manipulates attributes of the operating system's visual design and user interface to hide malicious activity [5]. *SlemBunk and Marcher* attack actively targets US financial institutions customers [6] and many more. Therefore, to ensure security and privacy of an Android user, detection of malicious apps becomes primary line of defence.

The main security mechanisms of Android are application sandboxing, application signing, and a permission framework to control access to (sensitive) resources. Android's security framework exhibits serious shortcomings:

- The burden of accepting app's permissions is assigned to the end-user who in general does not aware of the impact of prompted permissions on his privacy and security. Thus, malware can be installed on end-user devices such as sending of text messages to premium rate numbers or leaking of sensitive data in the background of running games.
- Android does not enforce any permission on sharing data through inter-component communication (ICC) mechanism (intents, shared preferences, and content providers). Therefore, ICC across apps expose possibilities of various threats including intent spoofing [7], component hijacking [7], confused deputy [8, 9], privilege escalation [10], collusion [11, 12], etc. In these attacks, the malicious app may send and/or receive sensitive data that it is not authorized to access and creates leakage paths.

The consequences of these attacks lead to privacy leak that occurs if there is a secret (without user consent) path from sensitive data as a source to statements that are sending this data outside the application or device, called sink. This path may be within a single component or across multiple components [11]. Thus it is necessary to perform an inter-component and inter-app analysis of applications.

With the growing use of Android and the awareness of its security vulnerabilities, the operating system is already providing various techniques to mitigate the risk of data theft and misuse. Many techniques have been proposed in academic research papers, both on the system level and on the application level. These research focus mainly on the following parameters:

- Least privilege principle plays a vital role in the classification of benign and malicious apps. The app with a lesser number of permissions is considered to be more secure.
- Mostly techniques propose the detection of data/privacy leakage paths present in an app. Almost no attention has been given to analyze multiple apps together.

Recent works have demonstrated that the main vulnerability comes from the fact that leakage paths are exacerbated by several applications that can interact to leak data using the inter-app communication mechanism [10, 13–15]. The aforementioned security risk could lead to the collusion attack resulting in privacy abuse [11]. The danger of malware collusion is that each colluding malware only needs to request a minimal set of privileges, which may make it appear benign under conventional screening mechanisms [11]. Therefore most of the state-of-the-art approaches, and the associated tools have long left out the security flaws that arise across the boundaries of single apps, in the interaction between several apps. The major challenge lies in the multi-app analysis is the search space posed by the possible combinations of apps.

1.1 Objectives

The purpose of this Thesis is to study state-of-the-art analysis tools, techniques and develop techniques to detect effectively and precisely Potentially Harmful Applications (PHA) on Android platform.

To achieve these goals, we analyze 'single app' and detect privacy leakage paths through 'multiple apps.' We focus on following objectives:

- Review the state-of-the-art techniques known for protecting user privacy and identify their shortcomings in doing so.
- Investigate and develop analysis techniques to complement existing single-app analysis approaches with improved analysis coverage. Implement techniques that effectively narrows down the search space of colluding apps candidates in order to perform a large-scale multi-app analysis.
- Develop a standard app dataset to compare and benchmark efficacy and scalability of methods dealing with collusion detection and make it available to the research community.

1.2 Motivation and Thesis Impact

Mobile devices provide an attractive attack surface for cyber thieves as the devices hold personal details and have potential capabilities for eavesdropping. Malicious hackers use the Android app as a tool to gain access to private information. The consequences of these attacks lead to damages that could be monetary or nonmonetary (loss of reputation, physical or mental pain or suffering).

The proposed analysis techniques presented in this Thesis will improve the detection accuracy of potentially harmful apps for Android platform. Furthermore, it will allow large-scale multi-app analysis to detect privacy leakage paths scattered across apps with reasonably good accuracy. Finally, the open-source dataset exhibiting collusion attacks can be used to verify efficacy and scalability of methods dealing with collusion detection.

1.3 Contributions

This Thesis is a step forward to protect the users' privacy by detecting potentially harmful apps that contain undesired data flow paths. We present techniques aiming this goal as well as a mechanism to cope with the challenges in intra and inter app(s) analysis. In summary, following are the motivations and contributions of this Thesis:

1. The proposed "DRACO" model is motivated by the fact that users want to analyze the app on their device. We propose an on-device analysis app that classifies malicious and benign apps based on the extracted features. It generates D-Score which is derived from the probability distribution of the app features towards maliciousness.

2. On-device analysis can produce the first level of warnings only, due to computational restrictions. To increase the code coverage and detection accuracy, server module needs to be equipped with off-device techniques. So, we propose a "SWORD" that encapsulates the semantics of Android apps using Asymptotic Equipartition Property (AEP) which are further quantified to detect the malicious apps.

3. It has been observed that capturing the behaviour of apps with regard to privacy is an important factor to differentiate malicious and benign apps in Android platform. So, we propose "FlowMine" that models the behaviour of an app in terms of sensitive data flow across execution path(s) of an app. The frequency of occurrence of a source-sink pair across a number of malicious and benign apps is obtained to determine if this pair can be used as a discriminant between malicious and benign behaviour.

4. Since, the state-of-the-art approaches focus on single-app analysis and hence failing to find security flaws arise across single apps boundary. We propose, "Intersection Automata based Model for Android Application Collusion" that represents apps communication through 'application automaton' and identify policies based on the violation of permission model through 'policy automaton.' The intersection of the two automata detects the presence of collusion.

5. To allow large-scale multi-app analysis by reducing the search space posed by the combinations of apps, we propose "SneakLeak+" that models app representing potential leaks only. It statically analyzes the reverse engineered intermediate code of each app, extract security relevant information, and represent the extracted information into a compact form suitable for formal verification. The formal analysis engine is used to experimentally verify the presence/absence of potential inter-app communication-based leakage.

6. Currently, there is no standard app dataset available to verify efficacy and scalability of methods dealing with collusion detection. Therefore, we developed 64 apps exhibiting collusion as our benchmark dataset, now, available as open-source at DroidBench [16] (which was verified by peer group before hosting). This is our contribution to the research community.

In this Thesis, our focus is to propose, design and implement analysis techniques for intra and inter app(s) for Android platform. These methods help in improving the detection rate of malicious apps and apps that can leak user's private information in collaboration.

1.4 Thesis Structure

The remainder of the Thesis is organized as follows. Chapter 2 provides an inside of Android and existing attacks due to its vulnerability. It evaluates state-of-the-art analysis techniques and their comparative study. Based on the existing reviews, we propose single as well as multi-app analysis techniques to improve the analysis coverage and detection accuracy. This is reflected in two parts of the Thesis. Part I focuses on the techniques proposed for intra-app analysis. In Chapter 3, we propose 'DRACO,' an on-device app analysis to classify malicious and benign apps. Chapter 4 presents 'SWORD' that encapsulates app's semantics using system-calls to categorize malicious/benign app. Chapter 5 explores 'FlowMine,' our proposed approach based on data flow analysis to detect apps that are maliciously leaking data to sensitive sinks. Part II of the Thesis focuses on inter-app analysis. In Chapter 6, we propose collusion detection technique for two apps based on the intersection of automata. Chapter 7 describes the design and implementation of a formal model for collusion detection for multiple apps. Finally, we conclude the Thesis with a pointer to the future direction.

Chapter 2

Analysis Techniques for Android App(s): A Review

Android apps that handle personal or sensitive user data by collecting information about the user (including personally identifiable information, financial and payment information, authentication information, phonebook or contact data, microphone and camera sensor data, and sensitive device data) [17] are considered as potentially harmful apps. We are witnessing an exponential upsurge in malicious apps, growing use of potentially harmful apps to create leakage paths that are scattered over several apps, banking frauds, etc. are targeting Android smartphones.

In this chapter, we discuss the Android security mechanism, app's composition, and communication mechanism. The chapter provides a comprehensive assessment of the strengths and shortcomings of the known state-of-the-art approaches to analyze and detect potentially harmful apps. This chapter also provides a base towards proposing techniques that can analyze single app as well as simultaneously analyze multiple apps to detect data leaks. We will interchangeably use app(s) for application(s) in this Thesis.

2.1 Android App Composition

Android applications are distributed as binaries in a regular format based on zip files with *.apk* as the file extension. It usually contains the following files and directories [18].

1. `Manifest file:` Manifest file is an XML configuration file (AndroidManifest.xml) one per app. It is used to declare various components of an application, their encapsulation (public or private) and the permissions required by the app. Android APIs offer programmatic access to mobile device-specific features such as the GPS, vibrator, address book, data connection, calling, SMS, camera, etc. These APIs are usually protected by permissions. For example, the Vibrator class, to use the `android.os.Vibrator.vibrate(long milliseconds)` function, which starts the phone vibrator for some milliseconds. The permission `android.permission.VIBRATE` must be declared in the app *Manifest* file.

2. `dex file:` A Dalvik executable (classes.dex), which contains the bytecode of the program.

3. `res directory:` Resources including string literals, their translations, and references to binary resources.

4. `layout directory:` XML layouts describing user interface elements.

5. `lib directory:` The directory containing the compiled code that is specific to a software layer of a processor.

6. `assets directory:` The directory containing applications assets, which can be retrieved by AssetManager.

An android app is composed of any combination of the following four components:

- Activities: The Android libraries consists of a set of GUI components specifically built for the interfaces of mobile devices, which have smaller screens and low power consumption. One type of such component is Activities that represent screens which are visible to the user;
- Services: They perform background computation;
- Content Providers: They act as database-like data stores;
- Broadcast Receivers: They handle notifications sent to multiple targets.

2.2 Android Security Model

Android security depends on restricting apps by combining app signing, sandboxing, and permissions.

2.2.1 App Signing

App signing is a prerequisite for inclusion to the official Android market (Google Play Store). App signature is the point of trust between Google and the third party developers to ensure app integrity and the developer reputation. Most developers use self-signed certificates that they can generate themselves, which do not imply any validation of the identity of the developer. Instead, they enable seamless updates to applications and enable data reuse among sibling apps created by the same developer [19].

2.2.2 Sandbox Environment

Android apps are executed in a sandboxed environment to protect the system, the user data, the developer apps, the device, the network, and the hosted applications, from malware [20]. Each app process is protected by an assigned unique id (UID) within an isolated sandbox. The sandboxing restrains other apps or their system services from interfering the app [21]. Apart from UID, a process may be assigned one or more group id (GIDs). For example, if an app has permission for a network resource (e.g., Bluetooth), the app process is assigned to the corresponding network access id. An app must contain a PKI certificate signed with the developer key. App signing procedure places an app into an isolated sandbox assigning it a unique UID. If the certificate of an app A matches with an already installed app B on the device, Android assigns the same UID (i.e., sandbox) to apps A and B, permitting them to share their private files and the *Manifest* defined permissions [22].

2.2.3 App Permission Model

App permission model regulates how applications access certain sensitive resources, such as users' personal information (e.g. phone book, gallery, etc.) or sensor data (e.g., camera, GPS, etc.). For instance, an application must have the $READ\_CONTACTS$

permission to read entries in a user's phone [23]. System permissions are divided into four protection levels. The two most relevant to this thesis are normal and dangerous permissions. Normal permissions require when the app needs to access data or resources outside the app's sandbox but involve very little risk to the user's privacy or the operation of other apps. For example, permission to set the alarm is a normal permission. Dangerous permissions are required when the app wants data or resources that involve user's private information or could potentially affect user's stored data or the operation of other apps. For example, the ability to read user's contacts is a dangerous permission [24].

2.3 Inter Component Communication (ICC)

Inter Process Communication (IPC) is known as Inter Component Communication (ICC) in Android [25]. It is the key features of Android programming model. It allows a component of an application to communicate and transfer data to another component of same or other application. ICC allows developers to leverage services provided by other applications. For example, a cab booking application can ask Google Maps for client's or driver's location information. This communication between applications can reduce developer's burden and facilitate functionality reuse. To support inter-component communication, there exist conventional methods viz Intents, Content Providers, and Shared Preferences.

2.3.1 Intents

Intents are the preferred message passing mechanism for asynchronous IPC in Android. ICC is widely facilitated through Intents [7]. Intents enable components of an application to invoke other components of the same or different applications. It is also used to pass data between different components through Bundles. It optionally contains destination component name or action string, category and data. The Android API defines methods called ICC methods that can accept intents and perform actions accordingly. For example, `startActivity(Intent)`, `startService(Intent)` etc. Based on the destination of ICC calls, they are categorized into two broad categories:

Implicit Intent

Implicit intents are used when the receiver of the intent is not fixed [26]. When an app wants to send the intent to all the registered components (registration is done using an intent filter in the *Manifest* file) within and across the installed apps. In the following, we present a sample code of implicit intent where `"com.example.msgSendFirst"` is the action string:

```
1 Intent intent = new Intent("com.exampke.msgSendFirst");
2 startActivity(intent);
```

Explicit Intent

Explicit intents are used when receiver of the intent is fixed. When an app invokes API call with explicit intent, the framework will deliver the intent to the component that is mentioned in the intent. A Sample code of explicit intent where `"this, LoginActivity.class"` is the address of the destination component:

```
1 Intent intent = new Intent(this,LoginActivity.class);
2 startActivity(intent);
```

2.3.2 Content Provider

Content Providers are used to transfer structured data across components of same or different apps. It stores information in tables like relational databases. To access or modify data in Content Provider, apps need `ContentResolver` objects. An app can also attach read and write permissions to the content provider it owns.

2.3.3 Shared Preference

Shared Preference is an operating system feature that allows apps to store key-value pairs of data. Its purpose is to be used to store preferences information. Apps can use key-value pairs to exchange information if proper permissions are defined when accessing and storing data.

2.4 Android Attacks

In Android, an attack is any attempt to expose, alter, steal or gain unauthorized access of users' personal or sensitive data or disable, destroy or gain unauthorized access of any resource without users consent. Android ensures security through its sandbox model, application signing and the permission model for managing IPC effectively and efficiently. In spite of these measures, Android is vulnerable to many security risks. According to the recent OWASP mobile security report [27], out of 91 reported security risks, 85 are recorded to be present in Android. This makes Android security a serious concern. Following are some of the common attacks in Android:

2.4.1 Code Injection Attacks

Malware authors are injecting irrelevant or independent code at runtime to alter the actual runtime sequence of events. This can evade existing malware detection techniques because an injected application can redirect program control to some other code and also can execute it. There are mainly three ways [28] to inject code into already running Android application *1)* using `DexClassLoader`, an Android app can invoke the classes' methods of any downloaded app during runtime, *2)* invoking API named as `createPackageContext`, an android app can load and invoke resources (images, files, and codes) of other app [28] and *3)* using OS shell [29].

2.4.2 Intent Based Attacks

We focus our attention on the security challenges of Android communication from the perspectives of Intent sending and receiving. In section 2.4.2.1, we focus on the Intent receiving and consider vulnerabilities related to receiving Intents coming from other applications. In Section 2.4.2.2, we consider how sending Intents to the wrong application can leak user information.

2.4.2.1 Intent Spoofing

Intent spoofing refers to a typical scenario where a vulnerable app has a component that expects Intent from Android framework or itself. If the component is exposed, then

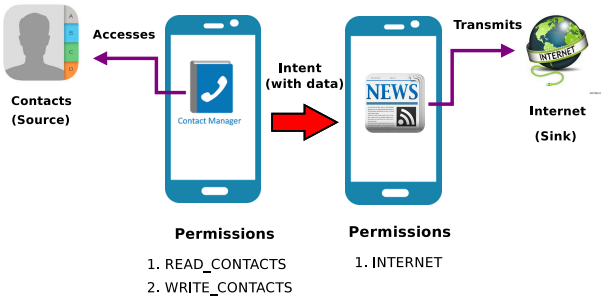


FIGURE 2.1: Collusion attack scenario

other malicious apps can send forged Intents, and then spoof this app in order to trigger misbehaved actions. We can classify the Intent spoofing to three subclasses: malicious broadcast injection, malicious activity launch, and malicious service launch [7].

2.4.2.2 Intent Hijacking

The Intent hijacking threat is illustrated when an Intent could not reach the intended recipient via an implicit ICC, and then it may be hijacked by an unauthorized app [7]. We can classify this threat based on the type of the sending component: broadcast receivers hijacking, activity hijacking, and service hijacking.

2.4.3 Collusion Attacks

Collusion refers to the scenario where two or more applications possibly (not necessary) developed by the same developer, interact with each other to perform malicious tasks. Figure 2.1, illustrates a collusion: there are two apps - `Contact Manager` and `News`. `Contact Manager` includes permissions `READ_CONTACTS` and `WRITE_CONTACTS`. On the other hand, `News` app possesses only `INTERNET` permission. They can communicate via Intent. If `Contact Manager` sends an Intent carrying contact information as a payload to `News` app which can transmit the information to the outside world using its permission. Then, in such scenario, sensitive information can be leaked without violating any security policies. The danger of malware collusion is that each colluding malware only needs to request a minimal set of privileges, which may make it appear benign to current state-of-the-art techniques that analyze one app at a time.

2.5 Review of Analysis Techniques

The analysis techniques based on the detection mechanism can be broadly categorized into static, dynamic and policy based techniques. This section explains each category, resources, and mechanism used in the analysis.

2.5.1 Static Analysis Techniques

Static analysis techniques act as a potential weapon for conducting the behavioural analysis of an application. It consists of examining and auditing the code without executing it [30]. Android apps are analyzed by inspecting the source code without actually running them. Static analysis extensively explores data flows in a program and subsequently detect paths through which information can be leaked.

2.5.1.1 Resources

To perform static analysis, information can be extracted from the *Manifest* file that includes the name of the package, list of components, list of permissions, version, information about intents and intent-filters used for communication, level of API and libraries required by an application for execution [31]. Dalvik executable file reveals information about the structure of an application and methods used by it. It is analyzed to detect potentially malicious actions (such as sending SMS to premium numbers, use of reflection or encryption, access to sensitive resources, etc.) [32]. Java libraries can be statically analyzed in order to obtain data flow summaries of an application. It can be useful to determine malicious flow in an application.

2.5.1.2 Mechanism

The mechanism employed to perform static analysis depends on the depth and purpose of analysis. Various static analysis techniques used by researchers such as taint analysis [33], dataflow analysis [34], entry point analysis [35] to name a few.

Taint analysis is also known as user-input dependency checking [33]. The concept behind taint analysis is that any variable altered by the user becomes tainted and is considered

vulnerable. The taint may flow from variable to variable during a course of operations, and if the tainted variable is utilized to perform some harmful operation, it becomes a breach of security. Taint analysis detects the set of instructions that are affected by user inputs. It helps in identifying sensitive information leakage. Data flow analysis determines the information flow between various components. It is the essential analysis need to detect leakage of sensitive data. For instance, for a variable, it can detect all the possible sources of a variable's value, i.e., where do values assigned to a variable come from, all the possible values a variable can possess, all the sinks where its value passes further, etc [36, 37]. Data flow analysis can be of various types depending on the context of analysis:

- Context-sensitive data flow analysis [38] examines target of a function call by focusing on calling context.
- Path sensitive data flow analysis [39] takes into account the branching statements. It analyzes the information obtained by the state obtained at conditional instructions.
- Flow-sensitive data flow analysis [40] considers the order of instructions in a program.
- Inter-procedural data flow analysis [41] takes into account the flow of information between procedures. It is achieved by constructing call graphs
- Intra-procedural data flow analysis [42] involves the flow of information within a procedure.

Entry point analysis [35] helps in determining where a program starts its execution. It is very challenging to identify starting point due to the use of callbacks and multiple entry points. Accessibility analysis [43] contributes in evaluating the likelihood of following a path between two components [30]. It helps in building reachability graphs showing the path followed through methods for execution of an application. A side-effect analysis is performed to compute which variables of a method are affected by its execution [37].

2.5.2 Dynamic Analysis Techniques

Dynamic analysis refers to the analysis of a program by executing it [30]. Android apps are examined and reviewed by actually executing them on real devices or emulators. Static analysis may miss some information that is generated during actual run, for example, network data stored in the heap memory during run time is not available before executing app, obfuscated strings are hard to recognize from decompiled codes, etc., therefore dynamic analysis is important to complement static analysis [44].

2.5.2.1 Resources

For dynamic analysis, the information about native code libraries, kernel parameters, CPU parameters, memory parameters, dynamically loaded libraries, components and processes currently running and invoked API calls, can be captured dynamically when the program is running [45].

2.5.2.2 Mechanism

Dynamic analysis mechanism includes System hooking [46], Taint analysis [13], Instrumentation [47], System call tracing [48], Debugging [49], Code emulation [50], to name a few.

System hooking involves altering or amplifying the functionalities of applications or components of the application, by anticipating function calls, events and transmitted messages between the components [51]. It is used to capture data flows, construct event ordering, record the parameters of passed messages and store values of run-time variables [52]. Dynamic taint analysis [53] starts by tainting the data that is initiated from untrusted sources, specifically user supplied inputs. Later, these tainted variables are stored, and whenever they carry sensitive data, they are tracked down to detect sensitive paths [50]. The term instrumentation pertains to the capability of monitoring or evaluating the performance of the product and interpreting errors [54]. Android apps are instrumented to monitor actions of specific components such as logging number of times a particular service is called, etc. It is achieved by injecting some code to keep a log of actions of specific components. In order to perform system call tracing, a system

call tracer is embedded into the system that logs the invoked interrupts or APIs as the program runs on the system [55]. In code emulation, the malicious code is executed on virtual machines with replicated CPU and memory management system, rather than real processor [55].

2.5.3 Policy Enforcement Based Analysis

Policy (a.k.a rule) enforcement based techniques make use of certain set of policies(rules) that are considered as normal or benign. These policies can be represented either in the form of regular expressions or any new policy language. The access of apps to any policy protected resource is verified against the predefined policy-set [56]. Verification can be done statically on the intermediate program code and can also be enforced at install-time or run-time. If the resource access adhered to the policy-set, it is considered benign. Any violation is referred as malicious behaviour [57]. The challenge posed by this defense mechanism lies in identifying, defining and maintaining the policy-set. It should not be very strict that may generate false-positives, but at the same time, it should not be too liberal to generate more false-negatives [58].

2.5.3.1 Resources

For policy-based analysis, extraction of information depends on the nature of the policy-set and where they are applied [59]. For example, if the policy set considers permissions and their corresponding API call then, *Manifest*, dex files and libraries are sufficed to extract relevant information. But, when policies are enforced at install-time or run-time, hooking or instrumentation need to be done. In the later case, relevant information can be extracted from system parameter like registers, CPU, etc.

2.5.3.2 Mechanism

The access control can be applied at various system abstraction layers viz. kernel-layer, middle-ware layer, and application layer [60]. The access controls are of various type viz. Mandatory Access Control (MAC), Discretionary Access Control (DAC), Role Based Access Control (RBAC), Context Based Access Control (CBAC) and Attribute Based Access Control (ABAC) [61, 62].

In MAC, whenever an app wants to access policy protected resource, Android kernel will verify the access to predefined rule-set. The access is allowed only if it is authorized. This rule-set is not modified by app or user. In DAC, user can define an access control list (ACL) on specific resources. These resources can be accessed when the owner provides permission. RBAC is based on the roles of an individual user. The user is assigning to different positions, with permissions to use the resources. The user can access sensitive data based on their assigned role. Till Android 5.1.1, once privileges are granted to the applications, they cannot be revoked. Despite that in many cases, whether the application gets a privilege or not depends on the user context and therefore CBAC comes into existence in Android. It has the capability to give privileges with dynamically granted or revoked to applications. In ABAC, granting privileges to the users is based on attributes which combine with the policies. Authorization relies on a set of operations that are determined by evaluating the attributes associated with the subjects, objects, and requested services.

In addition to these analysis techniques, there exist few hybrid approaches that benefit from the advantages of both static, dynamic and policy based techniques.

2.6 Comparative Study

Researchers have proposed various approaches for intra and inter-app analysis varying from static [30, 38], dynamic [44] to policy enforcement [56, 58] based techniques. Table 2.1 summarizes recent approaches under different criteria: (1) handled components, (2) handled Intents, (3) examines native code or not, (4) resolves reflection or not, (5) works on which code level, (6) conducts intra or inter-app analysis and (7) availability of the tool. We believe this helps the reader to examine all the differences in one glance.

Most of the proposed approaches, handle Android components viz. Activities(A), Services(S) and Receivers(R), whereas, Content Providers(C) are not handled by [7, 63–67] as shown in column (1) of the table. These approaches consider only Intents as a medium of communication. To access content providers, a unique resource identifier (URI) does not use the Intent. Therefore, Intent specific approaches fail to handle content providers. In particular, there are two approaches [68] and [69] that are not handling any components other than activities. In [68], the authors have mentioned that their approach

can be similarly extended for other components whereas [69] have built a prototype on activities and it is available commercially as a cloud service. In future the authors of [69] may extend their approaches to handle all the other components.

There are broadly two types of Intents viz. Implicit(I) and Explicit(E). All the proposed approaches can handle communication through Intents as they are the most popular medium of communication used in Android as shown in column (2) of the table. Although there is one approach [65] that is not considering implicit Intent. The reason narrated by the authors of [65] is that they do not want to increase false positives. In case of implicit Intent, the target is not fixed. If there are multiple receivers then at the run-time one of the receivers is chosen.

Column (3) of table 1 presents the capability of proposed approaches to handle native code. The native code refers to the code written in C/C++ and used by Android app libraries for low-level interactions with the underlying Linux kernel. The native code runs directly on the processor and hence not included in Dalvik executable that runs in Dalvik virtual machine. Almost all the approaches convert dex into some intermediate representation (IR) language but native code is not get converted into IR and hence, cannot be handled by many tools. Flowdroid [38] can handle very limited native calls as they defined some explicit rules for common invocation of native calls present in Java. Tools like [26, 68, 70] leverage Flowdroid for analysis and therefore can handle native calls partially.

The proposed approaches based on their ability to resolve reflection is depicted in column (4) of the table. Reflection is a language's ability to inspect and dynamically call classes, methods, attributes, etc. at runtime. It is a dynamic phenomenon, and hence it is very difficult for any static approach to handle it. Dynamic analysis approaches are needed to capture related runtime behaviour features to resolve reflection. If an API is called through reflection, it is passed as a parameter and hence become invisible for detection tools. Although some static tools like [26, 70, 71] can handle reflection partially meaning if the API calls are string constants, then they may be revealed otherwise if they are called through variable where it is obfuscated or encrypted, these tools cannot resolve such reflected calls.

Analysis tools based on the used intermediate representation (IR) for analysis are classified in column (5) of the table. Android *apk* file is converted to some IR prior to

	Proposed Approaches	Components Handled (1)	Intents Handled (2)	Native Code (3)	Reflection (4)	Code Level (5)	Inter-app Analysis (6)	Availability (7)
Static	MR-Droid [72]	⟨A S R C⟩	⟨E I⟩	No	No	Java Bytecode	Yes	-
	Detecting Inter-App Information Leakage Paths [67]	⟨A S R -⟩	⟨E I⟩	No	No	Java Bytecode & Smali	Yes	-
	Towards Automated Android App Collusion Detection [64]	⟨A S R -⟩	⟨E I⟩	No	No	Smali	Yes	-
	ICC Map [73]	⟨A S R C⟩	⟨E I⟩	No	No	Jimple/Source code	Yes	-
	IccTA [26]	⟨A S R C⟩	⟨E I⟩	Yes*	Yes*	Jimple	Yes^{+}	Open-Source
	Permission Flow [74]	⟨A S R C⟩	⟨E I⟩	No	No	Java Bytecode	No	-
	FUSE[71]	⟨A S R C⟩	⟨E I⟩	No	Yes*	Java Bytecode	Yes	Commercial
	AmanDroid [63]	⟨A S R - ⟩	⟨E I⟩	No	No	Java Bytecode	No	Open-Source
	DidFail [68]	⟨A - - -⟩	⟨E I⟩	Yes*	No	Java Bytecode	Yes	Open-Source
	ComDroid [7]	⟨A S R - ⟩	⟨E I⟩	No	No	Java Bytecode	No	Open-Source
Dynamic	IntelliDroid [65]	⟨A S R -⟩	⟨E -⟩	Yes	Yes	Java Bytecode	No	-
	IntentDroid [75]	⟨A - - -⟩	⟨E I⟩	Yes	Yes	Java Bytecode	Yes	Commercial
	TaintDroid [13]	⟨A S R C⟩	⟨E I⟩	Yes	Yes	Java Bytecode	No	Open-Source
Policy Based	DIALDroid [70]	⟨A S R C⟩	⟨E I⟩	Yes*	Yes*	Java Bytecode	Yes	Open-Source
	Intersection Automata Based Model for Android Application Collusion [66]	⟨A S R -⟩	⟨E I⟩	No	No	Java Bytecode	Yes	-
	FlaskDroid [60]	⟨A S R C⟩	⟨E I⟩	No	No	-	Yes	-
	XManDroid [76]	⟨A S R C⟩	⟨E I⟩	Yes	Yes	-	Yes	-

- A: Activity, S: Service, R: Broadcast Receiver, C: Content Provider E: Explicit Intent, I: Implicit Intent
- Yes*: The details are explained in section 2.6 Yes^{+}: If it is used with APKCombiner

TABLE 2.1: Comparison among state-of-the-art approaches

the analysis. There are four code levels on which analysis can be performed viz. Java source code, Java bytecode, Jimple, and Smali. Java source code can be analyzed because applications are written in Java language. The source is available only if the apps are open-sourced or self-developed. Android apps are compiled into Dalvik bytecode called Dex, which is executed in Dalvik virtual machine. For analysis, Dalvik should be converted to Java bytecode. This can be done by many *apk* to Jar converters like dex2jar [77], ded [78] and Dare [79]. Jimple is a simplified version of Java bytecode. It is a typed 3-address intermediate representation. It is used by Soot [80] which is a popular static analysis framework for Java. Dexpler [81] is a plugin for the Soot framework that translates Dalvik bytecode to Jimple. Smali is another IR used by very popular reverse engineering tool developed by Google named Apktool [82].

2.7 Summary

Android is a modern operating system for smartphones with expanding market share. The main security mechanisms of Android are sandboxing, signing, and a permission framework to control access to (sensitive) resources as discussed in Section 2.2. Android's security framework exhibits serious shortcomings as discussed in Section 2.4. The burden of approving application permissions is delegated to the end-user who in general does not care much about the impact of prompted permissions on his privacy and security. Thus, malware can be installed on end-user devices such as unauthorized sending of text messages or leaking of sensitive data in the background of running games as demonstrated in Section 2.4.

With the growing use of Android and the awareness of its security vulnerabilities, some research contributions have led to tools for the intra-app analysis of Android apps. Unfortunately, these state-of-the-art approaches and the associated tools have long left out the security flaws that arise across the boundaries of single apps, in the interaction between several apps. Based on the existing reviews in the chapter, we propose single as well as multi-app analysis techniques to improve the analysis coverage and detection accuracy. This is reflected in two parts of the Thesis. Part I focuses on the techniques proposed for intra-app analysis whereas Part II focuses on inter-app analysis.

In the next chapter, we will discuss intra-app analysis proposal *DRACO*, an on-device analysis technique to detect malicious apps.

Part I

Intra App Analysis

Chapter 3

On-Device Static Analysis

In the previous chapter, we discussed Android security model, its vulnerabilities, and existing analysis techniques. In this chapter, we propose, *DRACO*, an app-based analysis mechanism that utilizes the synergy of static, dynamic and machine learning based classification techniques. *DRACO* statically extracts following features:

- From *Manifest* file:
    1. Permissions
    2. Hardware Components
    3. Filtered Intents
- From *Dex* file:
    1. Suspicious API calls
    2. Restricted API calls
    3. Used Permissions
    4. Network Addresses

The proposed technique employs machine learning on the extracted features to quantify the risk posed by the app by providing a risk score. The goal of this chapter is to design an analysis technique capable of identifying risky apps (presence of restricted and suspicious API calls, unused permissions, suspected IP addresses, etc.) on-device.

3.1 DRACO: Overview

Android apps developed by naive programmers and posted without regular testing have exposed mobile devices to attacks like bugs, data-leaks, and confidential user data breach. Malware app detection, prevention, and mitigation are important concerns for anti-malware industries (like AVG, McAfee, etc.) and academia. Malware app detection approaches are static, dynamic and hybrid. Static app analysis [83] is performed by inspecting the complete code without executing it. Dynamic analysis [84] generates temporal or spatial snapshots of processor execution, memory, network activity, system call logs, SMSes sent, phone calls made, etc. to discriminate harmful app from normal. Both the methods, although complementary, can be used in combination to increase the code coverage.

Techniques proposed to analyze apps off-device cannot be replicated as it is on the mobile platform because of the concerns of limited memory, constrained processing, and restricted power availability. So, we propose, *DRACO* an Android app analysis mechanism that analyzes apps statically to inspire on-device analysis and extend code coverage, in combination with dynamic analysis. It has two modules, client module which is in the form of an Android app and gets installed on mobile devices and a server module to complement the analysis. The proposed methodology consists of following steps:

- Reverse engineer the *apk* to extract *Manifest* and other binary files and convert them to readable form
- Extract features of an application from the app's code and the *Manifest* file
- Classify using linear support vector machine classification algorithm
- Quantify risk posed by the app by generating a risk score called 'D-Score'

In the following subsections, we shall be discussing these steps in detail.

3.1.1 App Features Extraction

DRACO performs broad static analysis, gathering as many features of an application as possible from the app's code and the *Manifest* file. This step is performed by both

client as well as server module. Client module executes in a constrained environment and should complete in a timely manner. Therefore, it extracts features only from *Manifest* file whereas, server module extracts features from app's code. The overall step is illustrated in Figure 3.1 and is outlined as follows:

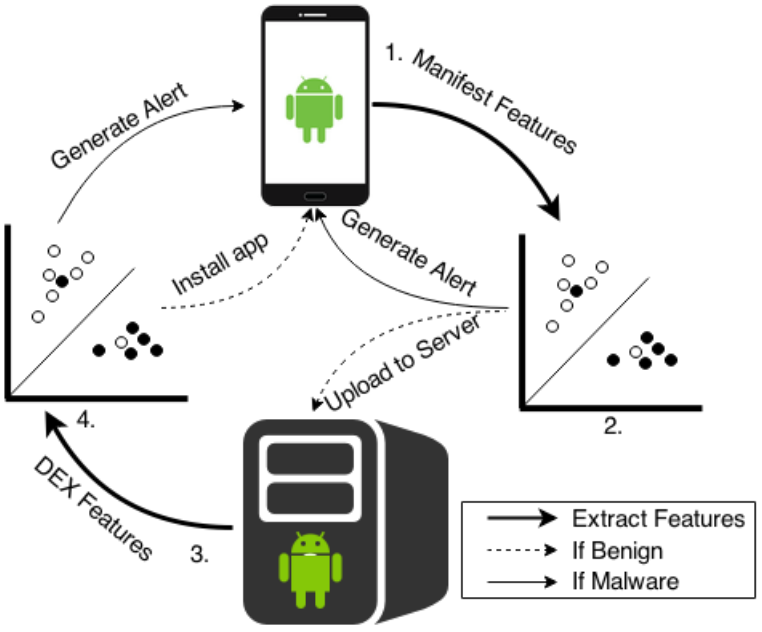


FIGURE 3.1: Static Analysis Phase

(a) ***Client Module***: Client module sits on device and user can choose *apk* file from the device for analysis. It statically inspects given *apk* and extracts different features from the app's *Manifest*. These features are explained below:

[D_1] **Requested Permissions**: In Android, permissions allow an app to access security-relevant resources. Permissions are actively granted by the user at the time of installation or the run-time (from Android 6.0) and are declared in the *Manifest* file of the app. Malicious software tends to request certain permissions more often like `SEND_SMS` permission or `READ_CONTACTS` permission. So D_1 is a feature set that consists of all the permissions mentioned in the *Manifest*.

[D_2] **Filtered Intents**: Intents are the messaging objects that facilitate communication between different app components like Activity, Service, Broadcast Receiver, etc. An intent-filter is an expression in an app's *Manifest* file that specifies the type of intents the component would like to receive. Malicious software tends to

listen some specific intent messages like `BOOT_COMPLETED`, `ACTION_SENDTO` etc. So D_2 is a feature set that consists of all the intent-filters mentioned in the *Manifest*.

[D_3] **Hardware Components**: Apps need to request hardware modules (like camera, touchscreen, GPS, etc.) if needed, in the *Manifest* file. Requesting access to specific hardware can cause security implications, as their combination may reflect harmful behaviour. For example, if an app has access to camera and network, it may leak pictures to the attacker.

(b) ***Server Module***: Server module sits on the server and accepts *apk* from the client module. The *apk* is disassembled into the smali format [85] (reverse engineered) to extract features from the code and identify any suspicious information or activity. These features are explained below:

[D_4] **Restricted API call**: In Android, there are some API calls that are executed only by the system apps (like REBOOT, DELETE_PACKAGES, etc.). *DRACO* searches the presence of such calls in the disassembled code as the use of such calls may indicate that the malware is doing privilege escalation in order to surpass the limitation imposed by the Android permission system. So D_4 is the feature set that consists of all the restricted API calls in the app's code.

[D_5] **Unused Permissions**: It may happen that the app is given a certain set of permissions which remains unused. In such cases, we can call the app as over-privileged. Thus, feature set D_1 becomes a ground for this set and further in our analysis we declare an app as suspicious. So D_5 is the feature set that consists of all the unused permissions.

[D_6] **Suspicious API calls**: Android permission model of security is an important measure to prevent unauthorized access to the sensitive resources. The app may access sensitive API calls without requesting their permission in the *Manifest* file. The use of such calls indicates that the app is under-privileged and can surpass the limitation imposed by the Android permission system. So D_6 is the feature set that consists of all the suspicious API calls from app's code.

[D_7] **Network addresses**: This set consists of all the IP addresses, hostnames, and URLs found in the disassembled code. Several organizations like Arbor Networks,

Google Safe Browsing APIs maintain and publish blocklists (a.k.a blacklists) of IP addresses and URLs of systems and networks suspected of malicious activities on-line. Many of these lists are available for free; some have usage restrictions. If these addresses are present in the app, it becomes suspicious. So D_7 is the feature set that consists of all such network addresses.

Client module produces a score called *DRACO* score (D-Score). It is the measure of the confidence with which the *apk* belongs to any class. It is calculated as the probability with which the app lies in malicious class. If the app is benign, the score is towards 0 otherwise it is towards 1. In addition to this score, we also explain why scanned *apk* was categorized as malware or benign. This is done using attribute weights. Learning happened off-device and learned model is embedded into the app for classification. This model gets updated on a regular basis.

3.1.2 Dynamic Analysis Phase

The second phase of *DRACO* is based on dynamic inspection of the app, in which the client module collects logs of CPU usage, memory usage and file operations of the running apps from the device and send it to the server. In this phase client module is responsible for collecting and uploading logs at regular intervals and server module is responsible for classification and detecting anomalies in behaviour. The output of the dynamic phase is in the form of push-notifications indicating any anomaly in running apps. This phase is illustrated in Figure 3.2 and outlined as follows:

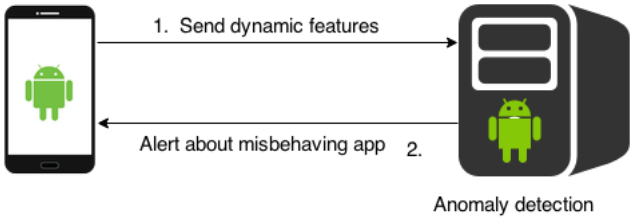


FIGURE 3.2: Dynamic Analysis Phase

(a) ***Client Module***: All the devices in which *DRACO* app is installed, become our users. For each user, we maintain the list of installed apps. Whenever the user runs *DRACO*, we fetch the running state of all apps. In running state, we consider memory consumption, CPU consumption and number of file operations of each app for that user. Whenever a user gets connected to the internet, it sends those collected logs to the server

module.

(b) ***Server Module***: This module considers per user analysis model to detect misbehaviour of any running app. The received logs become the input based on which, it classifies the app as malicious or benign using machine learning model. A warning is sent to the user for the malicious app.

3.1.3 Feature Vector Construction

This step focuses on embedding a feature vector (bit vector) using extracted *Manifest* features vector $D_M = D_1 \cup D_2 \cup D_3$ as shown in Table 3.1

Manifest Features	**Present Or Not**
android.hardware.wifi	0
android.hardware.telephony	1
android.hardware.camera	0
SEND_SMS	1
INTERNET	1
...	...

Table 3.1: Sample Manifest Features

Similarly, dex features vector D_D: As shown in Table 3.2, all are embedded in a joint vector space D_D such that: $D_D = D_4 \cup D_5 \cup D_6 \cup D_7$

Dex Features	**Present Or Not**
deletePackage()	1
SEND_SMS	1
INTERNET	1
setWifiEnabled()	0
sendTextMessage()	1
Runtime.exec()	1
...	...

Table 3.2: Sample DEX Features

Likewise, dynamic features matrix D_R: As shown in Table 3.3, CPU usage, memory usage and number of files accessed information is logged and embedded into a matrix D_R.

Client side static model is prepared from D_M, server side static model is prepared from

Running App	**Memory Usage**	**CPU Usage (in %)**	**Number of File Operations**
com.google.process.gapps	37592K	15	100
com.google.android.google quicksearchbox:search	41084K	33	30
com.android.systemui	32692K	3	101
android.process.acore	22380K	30	80
com.whatsapp	29904K	0	16
net.nurik.roman.muzei.muik	16484K	0	78
...	...	...	

TABLE 3.3: Sample Dynamic Features

$D_M \cup D_D$ and server side dynamic model is prepared from D_R. The state space of our extracted features is very large, in static phase feature set $D_1 \cup D_2 \cup ... \cup D_7$ counts to approximately 500. The state space will grow with the number of apps. So to predict the behaviour of an app with such huge state space is complex and inefficient. Therefore, we will infer such behaviour using machine learning techniques.

3.1.4 Machine Learning Algorithm

Benign and malicious samples are used to train the framework by recording behavioural information. The collected features are trained on supervised learning based model. There is an arsenal of machine learning methods that can be applied to learn a separation between malicious and benign apps, but only a few are capable of producing efficient results. We have chosen linear Support Vector Machine [86] for *DRACO* because of its highest detection accuracy rate and low false positive rate. It is a supervised learning model with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples as points in space, each known as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples to one or the other category. The machine learning model used 10-fold cross-validation to discriminate malicious app from benign. Table 3.4 illustrates machine learning classification results for 500 malware from Genome [87] and 500 benign apps from Google Play store [88]. Since the reported accuracy of SVM is more than Naive Bayes, we have done further experiments using SVM classifier.

Malware	**Benign**	**Feature Set**	**NaiveBayes (in %)**	**SVM (in %)**
Genome	Play Store	D_1	93.2	97.8
Genome	Play Store	D_2	94.3	93.2
Genome	Play Store	D_3	83.6	91.2
Genome	Play Store	$D_1 \cup D_2$	95	98.2
Genome	Play Store	$D_2 \cup D_3$	90.2	93.1
Genome	Play Store	$D_1 \cup D_3$	95.5	97.3
Genome	Play Store	$D_1 \cup D_2 \cup D_3$	**97.2**	**98.4**

TABLE 3.4: Accuracy of Machine Learning Models

3.2 Evaluation

All the experiments are done on DroidAnalyst server that is termed interchangeably as *DRACO* server in this chapter. The machine learning off-line models are also prepared on this machine. The configuration of DroidAnalyst server is: Processor i5 3.26GHz $\times$ 4 with 16GB of RAM. We have tested *DRACO* on ten different mobile devices with variation in brands, configuration, and Android OS version.

3.2.1 Experimental Results

Our dataset consists of 28,000 total sample apps out of which 18,000 are benign, and 10,000 are malware. Benign samples are collected from Play store using Akdeniz's google-play-crawler [89], and malware samples are collected from various sources like VirusShare [90], Genome Project [87], Contagion mobile malware minidump [91] and third party markets. The malware samples are categorized in 49 different families.

The training set consists of 22,000 samples and evaluation set consists of 6,000 samples. On an average time took to do 10-fold cross validation for 22,000 training samples was 90 seconds per model. The time required to do an off-device static analysis of the app sent by a user on DroidAnalyst is approximately 10 seconds. The detection accuracy of the test data-set is 94.8%. Figure 3.3 shows the interface of selecting an app on-device for analysis. Figure 3.4 illustrates on device analysis result with D-Score and the reasoning behind the score. Whereas, in Figure 3.5, the alert notifications indicates the anomaly found in running apps

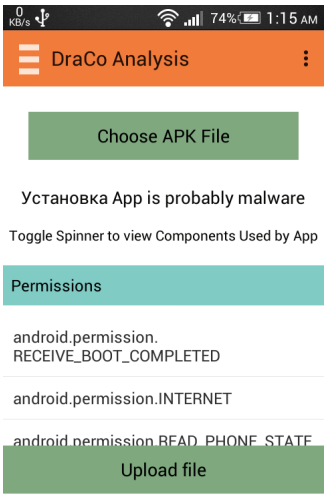


FIGURE 3.3: Analysis On Device

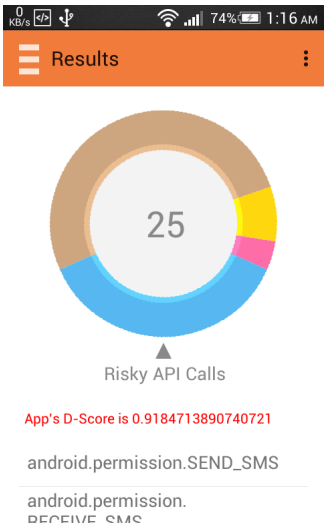


FIGURE 3.4: Analysis Result

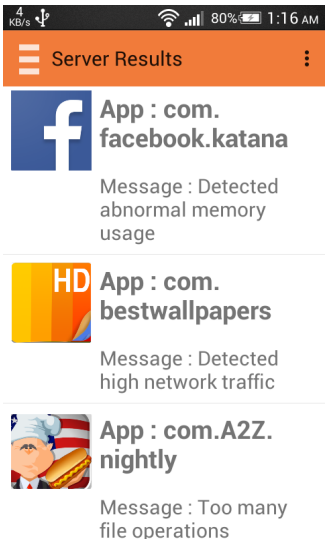


FIGURE 3.5: Server Results

3.2.2 Comparison with Existing Approaches

We tested *DRACO* with a wide range of malware families and it was found to perform better than many of the commercial anti-viruses. A comparison is shown in Table 3.5. We also compared our results with DREBIN[92] whose detection rate is given by Mobile Sandbox[93]. To determine results of top ten anti-malware, we upload samples from our test dataset on virus total[94] and record detection rate for each of them as shown in Table 3.5. DRACO reports highest accuracy in Full Dataset and VirusShare. On the MalGenome dataset, the anti-virus scanners achieve better detection rates as these samples have been public for a longer period of time. Hence, almost all anti-virus scanners provide proper signatures for this dataset [92].

3.3 Summary

In this chapter, we proposed and implemented a user-driven, on-device Android app assessment application, *DRACO*. Proposed application combines the synergy of static and dynamic analysis techniques to increase the code coverage and detection accuracy. Installation of this app does not need any root/super-user privileges. *Apk* file is scanned and a detailed analysis report is generated that classifies the app based on the features extracted from *Manifest* file and the code. To improve classification rate, *DRACO* performs dynamic analysis by using execution information of an app. Therefore, along with

Technique	**Full Dataset**	**VirusShare**	**MalGenome**
DRACO (Our Approach)	**94.80**	**93.90**	96.30
DREBIN	93.90	92.50	95.90
AntiVir	96.41	90.40	98.63
AVG	93.71	91.40	**98.90**
Bit-Defender	84.66	91.20	98.28
ClamAV	84.54	83.60	98.07
ESET	78.38	82.50	98.66
F-secure	64.16	80.40	96.49
Kaspersky	48.50	62.30	94.49
McAfee	43.34	34.50	84.23
Panda	9.84	18.6	23.68

TABLE 3.5: Detection rates (in %) of *DRACO* compared with other anti-malwares

the static features, it considers dynamic features like file operations, network operations and battery usage into consideration. Finally, reports and recommendations are sent to the user.

On-device analysis can produce the first level of warnings only. Due to computational restrictions, they are not capable of thorough app analysis. To increase the code coverage and detection accuracy, server module needs to be equipped with off-device techniques. In the next chapter, we present an off-device dynamic analysis method. The main limitation of the dynamic analysis methods is identifying all execution paths and check all these paths for the presence of malicious activity. Ensuring 100% code coverage is difficult especially when paths containing malware code are executed under certain user inputs and/or trigger conditions. Our proposal encapsulates an app as a collection of information-rich execution paths. Details of identification of such path and their usage in discrimination of benign and malicious app are presented next.

Chapter 4

Typical Path based Dynamic Analysis

The evolving malware can exhibit multiple behaviours at run-time. As discussed in Chapter 2, static analysis techniques are not applicable to apps that are encrypted, reflected or obfuscated. Dynamic analysis techniques are needed to combat these limitations. As a result, we capture the semantics of an app through dynamic analysis. System and hardware resources can be accessed only through system-calls. Majority state-of-the-art techniques rely on system-calls to model the running behaviour of apps. In this chapter, we propose Semantic AWare AndrOid MalwaRe Detector *(SWORD)* that encapsulates the semantically-relevant paths through the sequence of system-calls invoked by the apps. These paths are extracted from a sequential system-call graph derived from Android apps using Asymptotic Equipartition Property (AEP) inherited from information theory domain. These paths are further quantified to classify malicious apps from benign.

4.1 Encapsulating App Semantics using System-Calls

It becomes essential to capture the actual semantics (behaviour) of Android apps. It can be captured by logging system-call sequences because in Android system-call sequences are prevalent according to the family of malware. For example, *Plankton* is a malware

family that uses the update methodology for its propagation and hence make extensive use of system-call sequences like `socket()`, `connect()`, `sendto()`, `recvfrom()`, `socketpair()`. *DroidKungFu* is a malware family that aims to take root privilege `fchown32()`, `umask()`, `flock()`, `fork()`. Moreover, system-calls are independent of Android's compilation and running environment be it's Dalvik virtual machine (DVM) or Android runtime (ART). They provide a gateway to access system-level services and are required to instigate malicious attacks such as premium calls, downloading other malicious apps, transferring bank credentials and to name a few.

The majority of the dynamic analysis approaches make use of system-calls as these are only available gateways for an app's interaction with the operating system. Moreover, system-calls can be easily monitored by extending emulators (for example, QEMU) with *strace* and Monkey tool [95].

4.2 Information Theory

Our main objective is to define a metric that can quantify an app's behaviour. For this, SWORD employs asymptotic equipartition property (AEP) that is based on Shannon's entropy [96] widely deployed in the domain of information theory. AEP property states that "there are few paths of a graph that concentrates almost all information of the program under analysis".

4.2.1 Asymptotic Equipartition Property

In information theory, the *Asymptotic Equipartition Property*, *AEP* for short, states that if the system has independent, identically distributed (i.i.d. for short) random variables, then the observed sequences can be divided into two sets, *the typical set*, where the entropy is close to true entropy, and *the non-typical set*, which contains rest of the sequences. The typical set will determine the average behaviour of large samples with high probability [97] [98]. This is mathematically defined in Definition 4.1.

Definition 4.1. If $(X_i)_{i=1,n}$ are i.i.d. random variables, and
$\mathbb{P}r\left(X_1 = x_1, X_2 = x_2, \cdots, X_n = x_n\right)$ is the probability of observing the sequence

$\{x_1, x_2, \cdots, x_n\}$, then

$$-\frac{1}{n}\log P\left(X_1 = x_1, X_2 = x_2, \cdots, X_n = x_n\right) \rightarrow H\left(X\right) \text{ in probability,} \tag{4.1}$$

where, $H\left(X\right)$ is the entropy rate of X. Equation (4.1) is the AEP property [97]. According to this, in random processes, a small part of the sample sequences called a typical set of sequences contain almost all the relevant information. This property can be applied to only a few cases like i.i.d processes, stationary Markov chains and ergodic Markov chains [99, 100]. By identifying the typical set sequences, there is a significant reduction in the state space of the system under test.

4.2.2 Ergodic Markov Chain

Definition 4.2. A *Markov chain* is a discrete-time stochastic process $(X_t)_{t\geq 0}$ s.t. each random variable X_t takes values in a discrete set S, called *space state*, and for any s, s' and $s_0, s_1, \cdots, s_{t-1} \in S$,

$$\mathbb{P}r\left(X_{t+1} = s \mid X_t = s', X_{t-1} = s_{t-1}, \cdots, X_0 = s_0\right) = \mathbb{P}r\left(X_{t+1} = s \mid X_t = s'\right) \tag{4.2}$$

If the set S is finite then the chain is said to be *finite-state*.

Remark 4.3. Equation (4.2) is called *memoryless* property and it simply means that, as time goes by, the process loses the memory of the past.

The chain is characterized by the space state S and by its *transition matrix* $P = (p_{i,j})_{(s_i,s_j)\in S\times S}$, where,

$$p_{i,j} = \mathbb{P}r\left(X_{t+1} = s_j \mid X_t = s_i\right), \forall t \geq 0, \text{ and } \forall (s_i, s_j) \in S \times S.$$

Note that the transition matrix P verifies two properties: 1) its elements are all positive, and 2) each row sums to 1.

It is always possible to represent a finite-state Markov chain by a *transition graph* $G = (S, \tau)$ where S is the space state and τ corresponds to the transition matrix: for any

pair of states s_i and s_j in S, $(s_i, s_j) \in \tau$ iff $p_{i,j} > 0$. The graph G is, thus, an oriented weighted graph. Given $t \geq 0$, the *distribution* at time t of the Markov chain is given by:

$$\pi_s^{(t)} = \mathbb{P}r\left(X_t = s\right), \forall s \in S.$$

To characterize the chain completely, in addition to the space state S and the transition matrix P, one needs to specify the *initial distribution*:

$$\pi_s^{(0)} = \mathbb{P}r\left(X_0 = s\right), \forall s \in S.$$

Thus, knowing $\pi^0 = \left(\pi_s^{(0)}\right)_{s \in S}$ and P, allows to compute $\pi^t = \left(\pi_s^{(t)}\right)_{s \in S}$. Indeed:

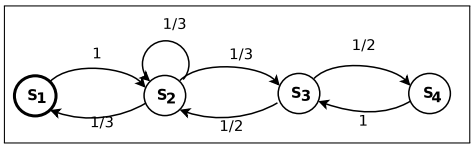
$$\pi^{(t)} = \pi^{(t-1)}P = \pi^{(0)}P^t, \forall t \geq 1.$$

Definition 4.4. A state s_j is *accessible* from a state s_i if the process, starting in state s_i, has a non-zero probability of reaching state s_j. This is equivalent to the following property in the transition graph G: there is an (oriented) path from s_i to s_j in G. The Markov chain is said to be *irreducible* if *all* its states are accessible one to the other. Equivalently, G has a single strong connected component.

- A state s is *periodic* with period d if d is the smallest integer s.t. $\mathbb{P}r\left(X_k = s \mid X_0 = s\right)$ for all k that are not multiples of d. In case $d = 1$, the state is said to be *aperiodic*.
- A state s is said to be *transient* if, given that the process starts in state s, there is a non-zero probability that it will never return to s. A state s that is not transient is said to be *recurrent*.
- Let s be a recurrent state and T_s be the first return time to s. If the expected value of T_s, given that the process starts in state s, is finite, then state s is said to be *positive recurrent*.

Now, we introduce the definition of ergodic Markov chains. This property is fundamental in the rest of this chapter.

Definition 4.5. A state s is said to be *ergodic* if it is aperiodic and positive recurrent. In other words, a state s is ergodic if it is recurrent, has a period of 1 and it has finite



(a) A transition graph

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 0 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) A transition matrix

FIGURE 4.1: Markov Chain Example

mean recurrence time. If all states in an irreducible Markov chain are ergodic, then the chain is said to be *ergodic*.

Definition 4.6. A probability distribution $\pi^* = (\pi^*_s)_{s \in S}$ satisfying $\pi^* P = \pi^*$, *i.e.*, $\pi^*_{s_j} = \sum_{s_i \in S} \pi^*_{s_i} p_{i,j}$ for all $s_j \in S$, is called a *stationary distribution* for the Markov chain $(X_t)_{t \geq 0}$.

Then, we have the following important theorem [99]:

Theorem 4.7. *An ergodic Markov chain $(X_t)_{t \geq 0}$ admits a unique stationary distribution π^*. Moreover, this distribution is also a limiting distribution, i.e.,*

$$\lim_{t \leftarrow \infty} \pi^t_s = \pi^*_s, \forall s \in S.$$

Example 4.1. *In this example, we illustrate the notions we introduced above. Consider a random walk in a set of sites s_1, s_2, s_3 and s_4. When the walker is in site s_1 (or in site s_4), it moves to site s_2 (or to site s_3). When he is in site s_2, he chooses uniformly at random (u.a.r. for short) to move to s_1, s_3 or stay at s_2. Finally, if he is in site s_3, he chooses u.a.r. one of the sites s_2 or s_4. At time $t = 0$, the walker is at site s_1. The random walk is then modelled by a Markov chain $(X_t)_{t \geq 0}$. This chain can be completely described by its space state $S = \{s_1, s_2, s_3, s_4\}$, the transition matrix P, and by the initial distribution $\pi^{(0)} = (1, 0, 0, 0)$, or by $\pi^{(0)}$ and the transition graph in Figure 4.1(a) and transition matrix in Figure 4.1(b).*

Assuming the initial distribution $\pi^{(0)} = (1, 0, 0, 0)$, it is then easy to compute the distribution at time $t = 1$: $\pi^{(1)} = \pi^{(0)} P = (0, 1, 0, 0)$, at time $t = 2$: $\pi^{(2)} = \pi^{(0)} P^2 = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0)$ and so on. This chain is ergodic. Indeed, state s_2 is ergodic since it is recurrent and aperiodic and hence all the states of the chain are ergodic also. It results

from this that there is a unique stationary distribution π^* and one can solve the system $\pi^* P = \pi^*$, $\sum_{i=1}^{4} \pi_i^* = 1$ to find out $\pi^* = \left(\frac{1}{7}, \frac{3}{7}, \frac{2}{7}, \frac{1}{7}\right)$.

4.3 Proposed Approach: SWORD

Our proposed approach is a combination of program semantics and a classification engine. We design our model on the top of QEMU [101] and Monkey tool [102] to automatically capture the run-time behavior of an Android app in terms of system-call sequences. We assume that acquired system-call traces represent the ergodic Markov chain and therefore we can apply the AEP concept to construct our semantic detector.

4.3.1 Implementation Details

SWORD's architecture is illustrated in Figure 4.2 and the broad steps involved in designing and implementing SWORD are the following:

1. *System-call Tracing*: Android apps are executed in the virtual environment to extract the system-call traces. We preserved the order of invoked system-calls as it will allow us to capture the actual behaviour of apps.

2. *Typify Program Behavior*: To typify the program behaviour, we construct Sequential System-call Graph (SSG) from the acquired traces by applying Markov property [103]. Assuming first and last invoked system call as the start and end node respectively, we compute all acyclic paths between them. Further, we apply AEP on each path and w.r.t an arbitrary value ϵ we create multiple ϵ-typical sets. Moreover, these typical sets will now be used to form our feature vectors.

3. *Statistical Analysis*: In this step, for each path in every typical set, we determine ALBF value. This ALBF value denotes the semantic quotient of each path and devise a reliable semantic detector. We apply histogram binning technique for forming our feature vector table (FVT).

4. *Train the Model*: To prepare the decision model, each of the formed FVTs is trained using a supervised learning algorithm.

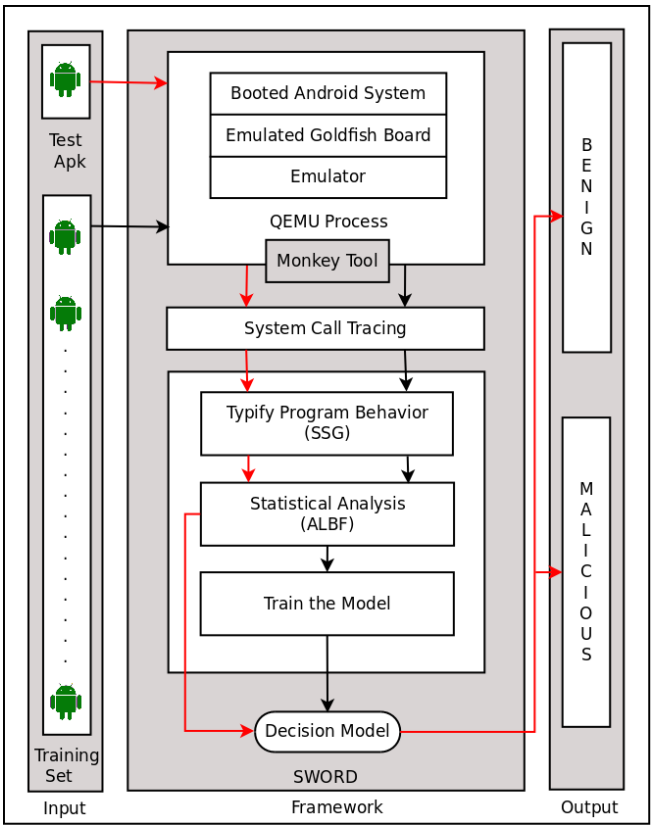


FIGURE 4.2: SWORD Architecture

4.3.1.1 System-Call Tracing

We extended QEMU [101] based Android emulator comes with Android SDK to capture behavior of the incoming *apk*. The behavior is captured by executing the app with randomly generated events that resemble user triggers. These events are generated by the attached monkey tool [95]. Monkey tool can generate pseudo-random events of clicks, swipes, touch screens, gestures, etc., to a real device or an emulator [104]. In our experiments, we configure Monkey to run 2500 events for each app. Whenever an event is triggered, the operating system will invoke corresponding system-call in order to serve the trigger. We log all these system-calls in the invoked order. We utilize system-call invocations and ignore their parameters, as it will enhance the scalability and sensitivity of our approach towards other system artifacts. We have considered 334 system calls which is the union of system-calls present in all the Android OS versions till Android 6.0. When we run each app (a total of 6000 traces), we observed that only 72 calls were invoked during execution. The reason for lower number of calls traced is that there are many services such as `chmod, break, umount, setuid, getuid, sched_setscheduler, setpriority, reboot, getpriority` to name a few

are invoked by system apps. Access to these services require root privileges. Therefore, the user apps (either benign or malicious) do not invoke system-services. Other services like `creat, link, unlink, chdir, lchown, stime,` etc. are used by very specific apps. Therefore, most of the system-calls are not invoked. Therefore, 72 system-calls that are invoked are referred as the 'frequent system-calls', and other remaining 262 system-calls are referred as 'rare system-calls'.

4.3.1.2 Encapsulating Program Behavior

In Android, only a single system-call will not suffice to capture the malicious behaviour [105]. But remembering the previous system-call in the sequences is important as it represents specific behavior. Ex. $\langle read(), write() \rangle$ represents reading from source and writing to sink, $\langle write(), ioctl() \rangle$ represents writing to Intent and sending Intent, $\langle Recvfrom(), SendTo() \rangle$ represents sending SMS to premium numbers, $\langle socket(), connect() \rangle$ represents connecting to socket, *etc.* To capture these semantics history-less nature of the Markov chain would be suffice. We create SSG by applying Markov property.

Definition 4.8. SSG: A Sequential System-call Graph (SSG) $G = (S, E)$ is a weighted directed graph, where S represents the set of all possible (distinct) system-calls. In Android, $|S| = 334$, *i.e.*, $S = (s_1, s_2, \cdots, s_{334})$ and each system-call $s \in S$ represents a vertex in SSG. E is the set of weights and defined as follows:

$$E = \left\{ E_{ij} | s_i \xrightarrow{\rho_{ij}} s_j;\ s_i, s_j \in S \right\},$$

where, ρ_{ij} denotes the transition probability from system-call s_i to system-call s_j. The transition probability ρ_{ij} is computed as follows.

$$\rho_{ij} = \frac{count(s_i \rightarrow s_j)}{\sum_{k=1}^{334} count(s_i \rightarrow s_k)},$$

where, $s_i \rightarrow s_j$ represents a transition from s_i to s_j. As in Definition 4.8, in graph G, the probability to transit from a state $s_i \in S$ depends only on s_i and not on any of its preceding states. Thus, SSG is satisfying Markov chain property as discussed earlier.

Also, the next state from s_i can be in any state of set S. Since the graph is irreducible to markov chain, we can say that all the paths in the graph G form ergodic markov chains and we can access all the states (syscalls) in this graph. With this we can also say that SSG will satisfy the following property.

$$\forall i \sum_{j=1}^{334} \rho_{ij} = \begin{cases} 0 & \text{if all entries in } i^{th} \text{ row is zero} \\ 1 & \text{otherwise} \end{cases}$$

.

Definition 4.9. Path: A path $P = \{s_1, s_2, \cdots, s_n\}$ is an alternate sequence of nodes and edges of G which starts from source s_1 and ends at destination s_n.

In our case, a path is the sequence of the system-calls starting from the source node (*i.e.*, the first system-call in the trace) to the destination node (*i.e.*, the last system-call in the trace). Since, we specified app's behavior in terms of system-call sequences, one iteration over a subsequence portrays the one of the behavior of an app. The cycle over a same subsequence reflects the repetition of a behavior, therefore to capture a particular behavior, cycle-free paths are sufficient. Moreover, the malicious apps do not repeat their malignant behavior in order to avoid detection. Therefore, we considered only cycle-free paths (all the intermediate nodes in the path are distinct) reachable paths from source node to destination node. Let $(S_i)_{i=1,n}$ be a walk in the graph G and let $P = \{s_1, s_2, \cdots, s_n\}$ be a path. In the sequal, we will denote $\mathbb{P}r\,(s_1, s_2, \cdots, s_n)$ the probability for the walk $(S_i)_{i=1,n}$, starting at $S_1 = s_1$ to follow the path P.

As mentioned in [99], we can apply AEP on SSG. According to AEP, in random processes, a small part of the sample sequences called typical set of sequences contains almost all the relevant information. To determine $\epsilon-$typical path set, we apply following property.

$$\left| \frac{1}{n} \log \frac{1}{\mathbb{P}r\,(s_1, s_2, ..., s_n)} - \lambda^* \right| < \epsilon,$$

where,

n is the total number of nodes in the path P,

λ^* is the maximal entropy rate of the *apk* under consideration,

ϵ is a real number greater than 0,

$\mathbb{P}r(P)$ is the path probability of path P, this is estimated simply by the frequency of the paths of a given length in all the path set. It calculates the probability of a walk in G follows path P, and it is given by:

$$\begin{aligned}\mathbb{P}r(P) &= \mathbb{P}r(S_1 = s_1) \cdot \mathbb{P}r(S_n = s_n|S_{n-1} = s_{n-1}, \therefore, S_2 = s_2|S_1 = s_1) \\ &= \mathbb{P}r(S_1 = s_1) \cdot \mathbb{P}r(S_n = s_n|S_{n-1} = s_{n-1}).\end{aligned}$$

The $\mathbb{P}r(S_1 = s_1)$ is the initial probability of node s_1 . The initial probability of a node s_1 is the probability of occurrence of s_1 among all the system-calls invoked in the execution trace. Maximal entropy λ^* of the *apk* is given by

$$\lambda^* = max \left\{ \lim_{n\to\infty} \frac{\log(T_n)}{n} \right\},$$

where, T_n is the total number of paths of length n in G. Now, we apply AEP to select typical paths from all the paths in the SSGs. This will significantly reduce the state space. Typical paths are capable of depicting the behaviour of apk because they are more probable than the other paths of the graph.

4.3.1.3 Statistical Analysis

This step aims to apply statistical analysis on the typical paths to train the model that can distinguish malicious and benign apps. To facilitate statistical analysis, we need numerical representation of the obtained typical paths. For this, we use Average Logarithmic Branching Factor (ALBF) as it converges a constant value for the entire problem space [106] *i.e.*, paths in our case. ALBF of a path P is calculated by the following equation:

$$\frac{\log B(s_1, s_2, ..., s_n)}{n-1},$$

where $B(s_1, s_2, ..., s_n) = \prod_{i=1}^{n} b(s_i)$ and b(s_i) is the branching factor of node s_i. ALBF metric of each path is used to construct the feature space as branching factor is a good indicator of semantic relatedness. Further, we gather paths that share similar characteristics using 'histogram binning' [107] as it can drastically reduce the feature space, thereby improving the system performance. Also, it can handle slight variation in the sequence of system-call traces. By considering only typical set of paths, we are

reducing the noise of feature vector space and hence can increase the accuracy and reduce the false rate.

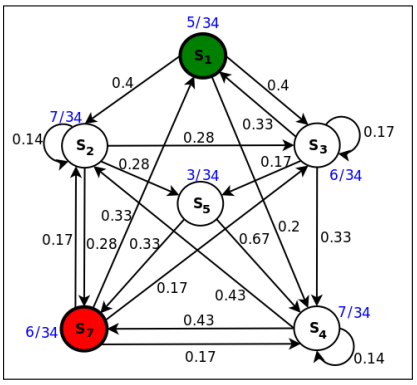
4.3.1.4 Train the Model

In this module, to classify malware and benign Android apps, we employ supervised learning techniques as we have labelled training dataset. Also, the supervised learning algorithm analyzes the malicious training data and produces an inferred function, which can be utilized for mapping new, unknown or zero-day malicious apps. On the other hand, the unsupervised learning methods are useful for identifying hidden patterns in an unlabeled dataset. In these methods, if the malicious apps are analysis-aware, these apps will not reflect their malicious behaviour and terminate their execution showing benign behaviour. In such scenarios, the unsupervised learning results in high false alarm rate. Therefore, we considered supervised learning in our approach. To this extent, we have used Waikato Environment for Knowledge Analysis (WEKA) [108]. In particular, we have used Random Forest [109] classifier. The advantage is, as forest building progresses, Random Forest generates an unbiased estimate of error [110]. Also, Delgado et al., experimented on 179 different machine learning techniques from 17 different families and applied them to 121 diverse datasets. They reported that Random Forest is the best classifier [111]. At the end of this module, we obtain a decision model that will be used for further classification of apps.

4.3.2 Demonstrating Example

In this section, we illustrate our approach using a simplified example of dummy ordered system-call trace. Let say there are in all 7 distinct system-calls $\{s_1, s_2, s_3,$
$s_4, s_5, s_6, s_7\}$ supported by an OS. Sample program trace ξ of a program based on this OS is $\xi = \{s_1, s_3, s_5, s_7, s_3, s_1, s_2, s_5, s_4, s_2, s_7,$
$s_2, s_2, s_3, s_3, s_1, s_3, s_4, s_7, s_4, s_4, s_7, s_1, s_2, s_5, s_4, s_2, s_7, s_1, s_4,$
$s_2, s_3, s_4, s_7\}$. The execution trace does not contain the system-call s_6, it means that s_6 is not invoked during the execution of this program.

The program behavior is captured through the Sequential System-call Graph (SSG) which is constructed from ξ. Each invoked system-call corresponds to one vertex of the



(a) SSG Graph

Node	s_1	s_2	s_3	s_4	s_5	s_6	s_7
s_1	0	2/5	2/5	1/5	0	0	0
s_2	0	1/7	2/7	0	2/7	0	2/7
s_3	1/3	1/6	0	1/3	1/6	0	0
s_4	0	3/7	0	1/7	0	0	3/7
s_5	0	0	0	2/3	0	0	1/3
s_6	0	0	0	0	0	0	0
s_7	1/3	1/6	1/6	1/6	0	0	0

(b) Transition Probability Matrix

FIGURE 4.3: SSG Example

SSG. As ξ has the sequence pairs (s_1,s_3), (s_3,s_5), (s_5,s_7), (s_7,s_3), (s_3,s_1), (s_1,s_2), (s_2,s_5) and so on, then edges are added from node 1 to 3, node 3 to 5, node 5 to 7, node 7 to 3, node 3 to 1, node 1 to 2 and node 2 to 5 and so on. SSG is a directed weighted graph as illustrated in Figure 4.3(a) and Figure 4.3(b) represents transition probabilities of every pair of nodes.

To typify program behavior, after constructing the SSG, all acyclic paths that are reachable from the source node (*i.e.*, the first system-call in ξ) to the destination node (*i.e.*, the last system-call in ξ) are extracted. Now, we select the candidates for typical path set. This selection is based on the frequency of paths for a particular path length.

Path Length	**Frequency**
2	2
3	4
4	5
5	2

TABLE 4.1: Frequency of paths corresponding to the path lengths

Table 4.1 shows the frequency of paths corresponding to the path lengths. To select potential candidates for the typical path, we can remove the paths that are less frequent.

In our example, the frequency of path length 2 and 5 is the 2, so we can remove them from being the candidates for typical paths. Therefore P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11} are the candidate path set as highlighted in Table 4.2.

Path Number	**Path Trace**
P_1	$s_1 \rightarrow s_2 \rightarrow s_7$
P_2	$s_1 \rightarrow s_4 \rightarrow s_7$
$\boldsymbol{P_3}$	$\boldsymbol{s_1 \rightarrow s_2 \rightarrow s_5 \rightarrow s_7}$
$\boldsymbol{P_4}$	$\boldsymbol{s_1 \rightarrow s_3 \rightarrow s_4 \rightarrow s_7}$
$\boldsymbol{P_5}$	$\boldsymbol{s_1 \rightarrow s_3 \rightarrow s_5 \rightarrow s_7}$
$\boldsymbol{P_6}$	$\boldsymbol{s_1 \rightarrow s_4 \rightarrow s_2 \rightarrow s_7}$
$\boldsymbol{P_7}$	$\boldsymbol{s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_7}$
$\boldsymbol{P_8}$	$\boldsymbol{s_1 \rightarrow s_3 \rightarrow s_4 \rightarrow s_2 \rightarrow s_7}$
$\boldsymbol{P_9}$	$\boldsymbol{s_1 \rightarrow s_3 \rightarrow s_5 \rightarrow s_4 \rightarrow s_7}$
$\boldsymbol{P_{10}}$	$\boldsymbol{s_1 \rightarrow s_2 \rightarrow s_5 \rightarrow s_4 \rightarrow s_7}$
$\boldsymbol{P_{11}}$	$\boldsymbol{s_1 \rightarrow s_4 \rightarrow s_2 \rightarrow s_5 \rightarrow s_7}$
P_{12}	$s_1 \rightarrow s_3 \rightarrow s_5 \rightarrow s_4 \rightarrow s_2 \rightarrow s_7$
P_{13}	$s_1 \rightarrow s_4 \rightarrow s_2 \rightarrow s_3 \rightarrow s_5 \rightarrow s_7$

TABLE 4.2: Candidate paths from the graph

Now, to extract the typical (semantically-relevant) paths, AEP is applied on the candidate path set that is reachable from the source node to destination node. Table 4.3 gives the AEP property value and ALBF value for each path in the candidate path set.

Path	**Probability of Path**	**AEP Value**	**ALBF Value**
P_3:1-2-5-7	$\mathbb{P}r(P_3)$:0.0056	1.203	30.99
P_4:1-3-4-7	$\mathbb{P}r(P_4)$:0.0084	1.057	34.33
P_5:1-3-5-7	$\mathbb{P}r(P_5)$:0.0033	1.394	30.26
P_6:1-4-2-7	$\mathbb{P}r(P_6)$:0.0018	1.612	35.07
P_7:1-2-3-4-7	$\mathbb{P}r(P_7)$:0.0012	1.3605	32.77
P_8:1-3-4-2-7	$\mathbb{P}r(P_8)$:0.0012	1.3605	32.77
P_9:1-3-5-4-7	$\mathbb{P}r(P_9)$:0.0028	1.1161	29.71
P_{10}:1-2-5-4-7	$\mathbb{P}r(P_{10})$:0.0048	0.09605	30.27
P_{11}:1-4-2-5-7	$\mathbb{P}r(P_{11})$:0.0012	1.3605	30.27

TABLE 4.3: AEP and ALBF value of candidate paths

Till now we have constructed SSG from the program trace ξ and identified all acyclic paths that are $P_1, P_2, \cdots, P_{13}$. Based on the frequency of paths per path length, we

have selected some candidates for the typical path. Then we have calculated the value of AEP property and ALBF value for each path. The minimum and maximum value of AEP property value are 0.09605 and 1.612 respectively. To determine the typical path set, we select the ϵ from the range 0.09605 to 1.612. For each selected ϵ, we select paths whose value is less than equal to selected ϵ value and call that set of paths as typical path set. For example, from Table 4.3, for $\epsilon = 1.4$, the typical path set contains the paths P_3, P_4, P_5, P_7, P_8, P_9, P_{10} and P_{11} because for these paths AEP value is less than 1.4. Figure 4.4 illustrated typical paths (blue colored). We can vary ϵ within the specified range and can evaluate the set of typical paths corresponding to that ϵ value.

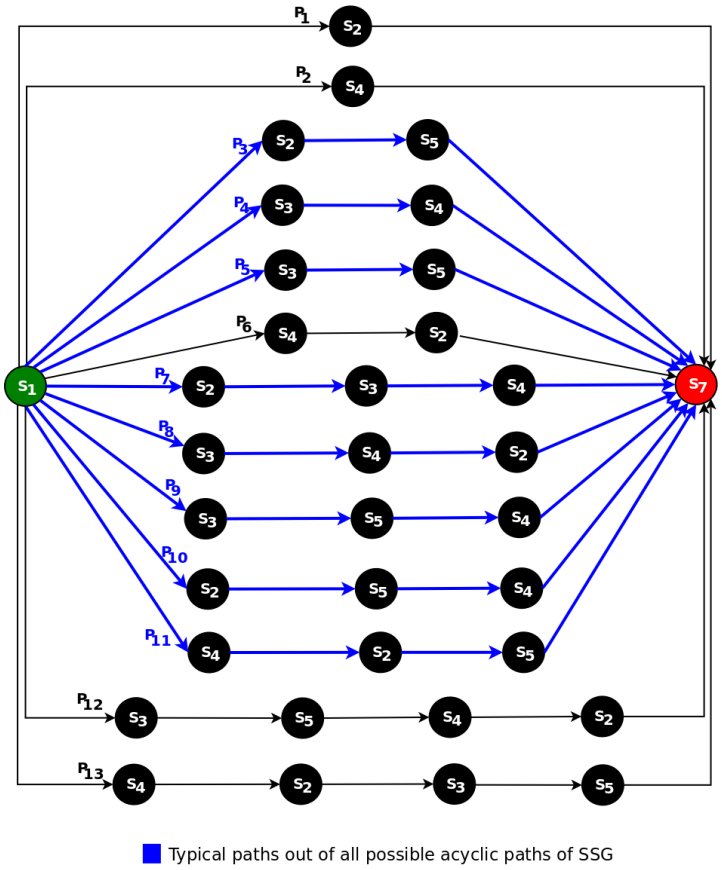


FIGURE 4.4: Typical Paths

We have typified the program behaviour from the system-call traces to typical path sets based on some specific ϵ value. Now to select the best possible ϵ value and construct the model, we represent typical paths in the numeric model called feature vector. Based on ALBF values range, we divide paths into certain bins that contain the frequency count of those typical paths that belong to that bin. The data in these bins will be our feature vector. For example, let the bin size is 1, that's means bins are placed at an interval

of 1. Suppose there are 8 such bins, $\{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\}$. For $\epsilon = 1.4$, the typical path set is $\{P_3, P_4, P_5, P_7, P_8, P_9, P_{10}, P_{11}\}$. So paths P_5, P_9, P_{10}, P_{11} belong to b_1, P_3 belongs to b_2, P_7, P_8 belong to b_4 and P_4 belongs to b_5. So the feature vector will $\langle 4, 2, 0, 2, 1, 0, 0, 0\rangle$. Once this feature vector is constructed, we train our classification model using Random Forest algorithm. Through experimentation, we select the best value of the ϵ that differentiates malicious apps from benign apps. Therefore we select the ϵ, where accuracy is higher and false rate is lower.

4.4 Performance Evaluation

All the experiments are carried out using benign and malware executable samples. To conduct these experiments, we have used Intel core with 16 GB RAM on OS Ubuntu 14.04. The major challenge lies in multiple entry points of Android. In Android, there does not exist a single entry point. Therefore each execution may lead to different sequences of system-call traces. For this reason, all the apps were executed on the emulator, *i.e.*, virtual environment three times. We considered each trace as a different app.

4.4.1 Dataset Preparation

Experimental Dataset consists of a total of 2000 android apps. Out of these 1000 are benign, and 1000 are malicious apps. To maintain the diversity the samples and avoid biases of the approach towards a single source, we have collected malicious dataset from four different sources *viz.* Genome Project [87], Inter-Component Communication Repository (IccRE) [112], New Malware Families 2015, Contagio Minidump [91]. To form our dataset, 250 apps from each of four sources are collected. The experimental dataset also contains 1000 benign apps that are chosen randomly from 12,000 apps of Google Play Store. To check the non-maliciousness of these apps, all apps are scanned on the VirusTotal [94]. VirusTotal is freely available online web service to scan the file or URL to check that it is malicious or not.

4.4.2 Approximate All Path Computation

We construct SSG using system-call trace of an Android app. The SSG is used to determine the full acyclic path form the S_{start} node to S_{end} node. S_{start} is the first system-call invoked and S_{end} is the last system-call invoked during the app execution. To evaluate full path between two nodes in a graph is an NP-Complete problem [113]. But the constructed SSG is sparse. The reason is that although there are 334 documented system-calls used in Android while analyzing 2000 real-world apps, we observed that only 74 system-calls are frequently used by most of the apps. In the graph, states are represented by system-calls. We are not facing any issue of state space explosion because system-calls are fixed. There is a difference between API-call and system-call. An app can call any number of API-call, but the set of system-calls corresponding to API-calls are limited.

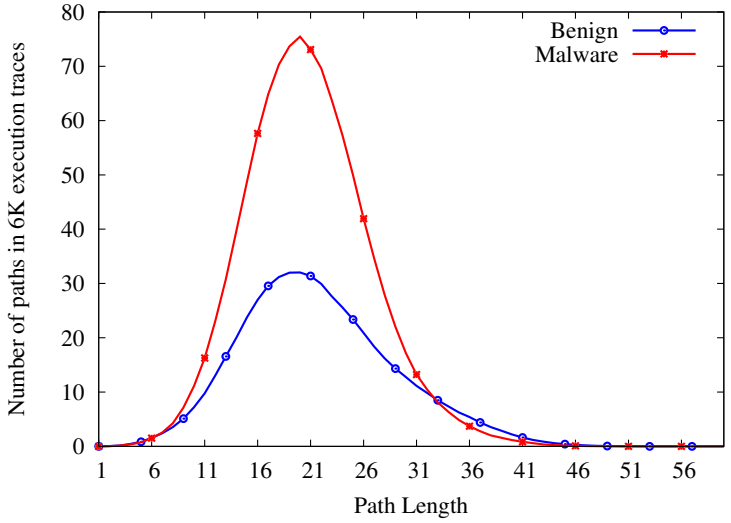


FIGURE 4.5: Path distribution w.r.t. to length in the malware and benign samples.

We have statistically obtained the evenly (normally) distributed path distribution for benign and malicious apps as illustrated in Figure 4.5. We utilized this distribution for observing two important points, 1) the path distribution is evenly (normally) distributed, therefore we can infer that our dataset is not biased for specific path-lengths, and 2) the path distribution is less for boundary path lengths (higher and lower). Therefore to discriminate malicious and benign apps, we can ignore the boundary path-lengths. We observe that there is a variation in paths of benign and malware samples in the path length ranging from 10 to 28. This variation in the number of paths can be used

for differentiating malware and benign apps. Therefore, we compute candidate paths instead of all paths to save path computation time for all the samples. So in this way, we approximate our all path computation phase.

4.4.3 Detection Accuracy

After extracting candidate paths, we apply AEP on each path with different values of ϵ ranging from 0.05 to 7.00. In this particular range, we construct various typical sets. We aim to determine a proper typical set that differentiates malicious apps from benign apps. All typical sets are transformed into FVTs and trained with 10-fold cross-validation. Table 4.4 shows the true positive rate (TPR), false positive rate (FPR), true negative rate (TNR), false negative rate (FNR), and accuracy of each of the FVTs constructed. In our case, malware is a positive class, and benign is the negative class. Therefore, TPR (TNR) is the true positive (negative) rate that indicates the rate by which malware (benign) is classified as malware (benign). FPR (FNR) is the false positive rate that indicates the rate by which malware (benign) is misclassified as benign (malware). To select the best value of ϵ, we did thorough experimentation with different values of ϵ with an increment of 0.5 in the previous value of ϵ as shown in Table 4.4. The lower values of ϵ do not capture the semantically rich paths, and therefore, we get very low accuracy for small values of ϵ. At higher values of ϵ, typical-sets contain semantically-rich paths as well as few paths that are also common to benign samples. Therefore, the FVTs formed using these sets result in low accuracy. We select the value of $\epsilon = 2.94$ where accuracy is higher and false rate is lower in order to differentiate malicious apps from benign apps. Also, the achieved accuracy of 94.2% indicates that the typical set at ϵ value 2.94 produces diverse path distribution across the feature vector table. Therefore, it can be marked as semantic-threshold for the testing phase. To obtain a semantic-threshold value, we have performed exhaustive experimentation for ϵ minimum range to maximum range. The lower values of ϵ do not capture the semantically rich paths, and therefore, we get very low accuracy's for small values of ϵ. At higher values of ϵ, typical-sets contain semantically-rich paths as well as few paths that are also common to benign samples. Therefore, the FVTs formed using these sets result in low accuracy.

ϵ	**TPR**	**TNR**	**FPR**	**FNR**	**Accuracy**
1.00	50.0	90.0	10.0	50.0	70.0
1.50	62.0	85.0	15.0	38.0	73.0
2.00	85.0	71.5	28.5	15.0	78.3
2.50	87.0	82.0	18.0	13.0	84.5
2.70	96.5	87.6	12.4	03.5	92.1
2.86	94.5	90.8	09.2	05.5	92.7
2.90	93.3	92.3	07.7	06.7	92.8
2.94	**95.8**	**92.6**	**07.4**	**04.2**	**94.2**
2.98	97.3	89.6	10.4	02.7	93.5
3.20	95.0	90.8	09.2	05.0	92.9
3.50	91.0	83.9	16.1	09.0	87.5
4.00	89.0	80.5	19.5	11.0	84.8
4.50	93.3	78.0	22.0	06.7	85.7
5.00	91.3	80.5	19.5	08.7	85.9
5.50	90.5	81.0	19.0	09.5	85.8
6.00	89.3	81.7	18.3	10.7	85.5
6.50	89.8	81.9	18.1	10.2	85.9
7.00	89.8	81.9	18.1	10.2	85.9

TABLE 4.4: Detection Rates in %

4.4.4 Comparison with Existing Approaches

To show the efficacy of our approach, we compare with existing work from the approaches that claim to detect malicious Android apps using semantic features. Table 4.5 summarizes the evasion capability and accuracy w.r.t. to the dataset size used by the approaches. MamaDroid [114] proposal relies on capture app's behaviour through the sequence of API calls invoked by the app as Markov chains. This sequence acts as features to classify benign apps from malicious. Similar to ours, their intuition also relies on the fact that malware exhibit different operations in different order than benign apps. They experimented with a huge dataset and reported detection accuracy ranging from 75%-86%. However, the proposed approach is different from our in terms of analysis technique and feature attribute selected for classification. MamaDroid is a static analysis technique and hence fails to capture the runtime context or malicious code that is executed. They select API calls as the feature attribute for classification that can be easily evaded by the self-defined packages that look similar to Android's, Google's or

Proposed Approach	**# Feature**	**Feature Type**	**Dataset Apps**	**Type & Origin of Dataset**	**Training/ Testing Distribution (in %)**	**Detection Accuracy**	**Handle Evasive apps**
MamaDroid [114] (Static)	64 - 116,281	API calls	44000	8500B (PD [115]+GPS) 35500M (DB+VS)	66.7/33.3	75%-86%	N
DroidSeive [116] (Static)	634-859	Resource Entropy + Cryptographic Libraries	9301	8041B (McGW) 1260M (GM)	67/33	92.38%	Y
CrowDroid [117] (Dynamic)	60	SysCalls	60	50B (SM) 10M (SM)	-NA-	85%-100%	N
ANDect [118] (Dynamic)	-NA-	API calls + SysCalls	1000	750B (GPS) 350M (CG)	90/10	88%	N
DroidScope [119] (Dynamic)	-NA-	-NA-	7	DroidKunfu Family DroidDream Family	-NA-	-NA-	N
CopperDroid [120] (Dynamic)	-NA-	SysCalls	2900	1365 (McGW) 1612M (CG+GM)	-NA-	60%	N
DroidScribe [121] (Static)	110-254	SysCalls+ API Calls +PER+SMS +USR	5246	5246M (DB)	-NA-	84%-94%	N
SWORD (Our approach) (Dynamic)	93	SysCalls	2000	1000B(GPS) 1000M (GM+ IccRE+CG+NMF)	70/30	94.2%	Y

TABLE 4.5: Comparison with related state-of-the-art approaches
PD:PlayDrone, GPS:Google Play Store, McGW: McAfee Goodware, GM: Genome Project, SM:Self-made, CG:Contagio Minidump, SysCalls: SystemCalls

Java's packages. Whereas, we considered system-calls as the feature attribute that is nonbypassable and hence giving an edge to our approach.

DroidSeive [116] is a static analysis based classifier to identify malicious app and its family. Their main focus is on capturing obfuscated malware. When trained on samples from malware genome [87] and McAfee goodware apps, the reported detection accuracy is 92.38%. The accuracy decreases with time when new obfuscation techniques come into play. The approach carries limitations of static analysis techniques as well.

Other approaches like CrowDroid [117] shows 100% on self-made apps. DroidScope has very small dataset size, CopperDroid has not reported their accuracy. DroidScribe, ANDect have a comparable size of the dataset and SWORD significantly outperforms them.

Our work is inspired by Smita et. al [122] approach that detects Windows based desktop malware. Although, due to difference in the architectural platform from desktop OS Windows to Mobile OS Android, there are significant modifications in the proposal. For example, the file format used in Windows is executable, and the execution has a single

entry point, whereas in Android the file format is *apk*, and it has multiple entry points. Each time when we run the app, it may exhibit different traces. It is a huge challenge to capture the actual behaviour of the app with multiple entry points. Another major challenge in Android is to identify the thin line between the behaviour of the malicious and benign app. For example, in Android, each app will get connected to the remote server for its update procedure. So we cannot consider it as malicious behaviour whereas, in Windows, this can be easily considered as malicious.

In our approach, we achieved 94.2% due to the diverse nature of our feature vector. We utilized several features to classify malicious behaviour. To select relevant features, we applied AEP property under which few important metrics were computed for different values of ϵ. For each value of ϵ, we obtain a different set of features (path sequences). Then we employed learning algorithm on the obtained features. We observed the true and false alarm rate. Our experiments showed that on $\epsilon = 2.94$, we could discriminate malicious and benign apps with reasonable accuracy.

4.4.5 Resiliency towards System-call Injection Attack

We used system-call injection attack to measure the robustness of the proposed approach. The system-call injection attack is a variant of a code-injection attack. The code injection refers to an attack where in an attacker is able to inject and execute malicious code by exploiting vulnerability (OS or software). The semantics of modern operating system (Android) prevents apps from having any outside effect unless they invoke system call. Therefore, injected malicious code cannot damage the system without invoking system call [123]. We, in our approach, considered the scenario where a malicious code is injected for modifying the system-call sequence of a already running malicious app in order to hide its malicious intent. For this experiment, we have selected a small set of 100 malware samples. As reported earlier, we divide system-calls into two categories, *i.e.*, rare system-calls (that were not invoked by any of the benign and malicious apps) and frequent system-calls (that were invoked during benign and malware app execution).

The experiment is conducted in two parts. In the first part, we inject rare system-calls into malware traces. In the second part, we insert frequent system-calls (randomly selected from benign traces). For both the parts, we construct feature vector table and compute the detection accuracy. We consider the trace-length of malware samples as a

parameter to decide the number of calls to be injected. We inject 10%, 20%, $\cdots$, 100% of malware trace-length (which varies approximately from 1K to 29K) into malware traces. Further, we closely monitor the fall in detection accuracy with these injection rates. Figure 4.6 shows the detection accuracy for rare and frequent system-calls.

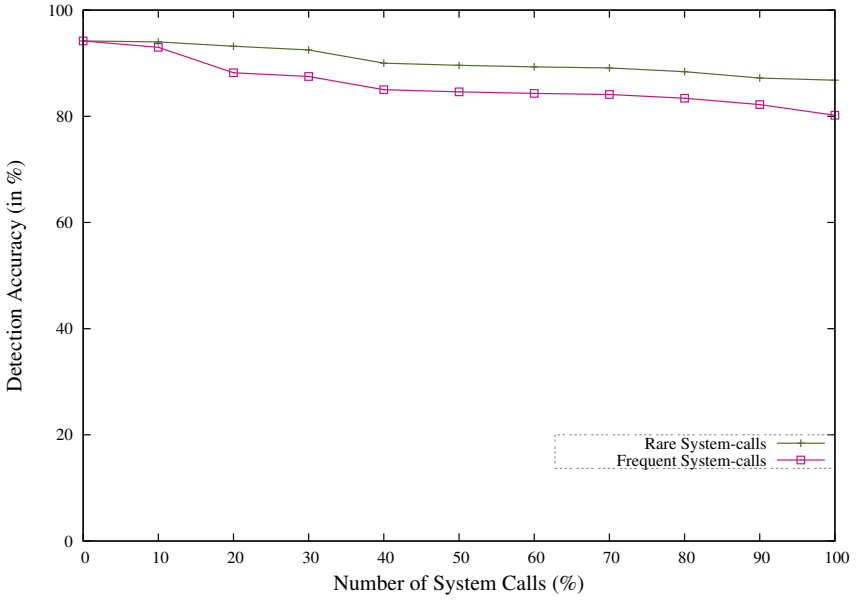


FIGURE 4.6: Injection Detection Accuracy

It is observed from Figure 4.6, that upto 30% of call injection the detection accuracy remain almost similar to 0% injection in case of rare system-calls. On the other hand, in case of frequent system-calls, our approach sustains upto $\sim$10%. This difference in detection accuracy is due to the nature of calls injected. The rare calls do not modify path distribution, therefore, our approach can sustain upto 30%. But, in case of frequent system-calls, the injected calls represent the benign sequences, therefore, the path distribution is modified and resulted in less sustainability.

The discriminating component of our approach is neither a sequence of system-calls nor a feature space linearly derived from these sequences. The discriminating component of our method is composed of the ranges of ALBF values. As these values are accumulated in bins, our feature space is non-linearly related to the sequence of calls. Multiple semantically relevant paths imply different subsequence of calls being used in the construction of feature space. Modification in one path shall not impact the performance of the proposed model. Only large modifications in transitions of all semantically relevant

paths will affect our model. The modification is complex as the attacker needs to identify all semantically relevant paths and to modify the path sequences in a way that it substantially modifies ALBF bins. Therefore, our approach is resilient to the injection attack.

4.5 Summary and Limitations

In this chapter, we proposed and implemented SWORD, a malware analysis technique that relies on capturing the semantics of an Android app during its execution in a virtualized environment. This semantics is further quantified to classify malicious and benign apps. System-call injection is one of the major evasion techniques used by attackers to alter control/data flow of the program at runtime. Our experimental results indicate resiliency of our proposed approach towards such attacks.

With almost universal digital convergence, users are extending functionalities of their devices by installing apps from various developers and vendor in an open ecosystem. These apps may misuse the sensitive information stored on the phone or obtained from the sensors to violate user's privacy. There is a need to analyze the actual behaviour of apps with regard to privacy. The static data flow analysis is a means for automatically enumerating the data flow inside a program. Our next work will focus on analyzing Android apps via data flow.

Chapter 5

Data Flow based Privacy Leakage Analysis

In the previous chapter, our analysis is based on encapsulating semantically-relevant paths to identify the behaviour of any app. Capturing the behaviour of apps with regard to privacy is an important factor to differentiate malicious and benign apps.

In this chapter, we propose FlowMine that models the behaviour of an app in terms of sensitive data flow across execution path(s) of an app. Such behaviour can be captured by identifying the data flow path from a data source to a data sink, where 'source' is a non-constant data that marks the beginning of the path, and 'sink' is the resource where the data is destined to. The frequency of occurrence of a source-sink pair across a number of malicious and benign apps is obtained to determine if this pair can be used as a discriminant between malicious and benign behaviour. Each source-sink pair is assigned a rank, which is indicative of its discrimination capability. Our method is a static approach, and it is assumed that the app is free from any encryption/compression/reflection in its code.

5.1 Data Flow in an App

In Android, an app may have the capability to access data stored in the phone (including identification information, financial or payment information, contact details, gallery pictures, etc.) or access data from sensors (including GPS, microphone, camera, etc.).

It may also have the capability to transmit the data outside the device through SMS or Internet.

Data flow is the path of the data from source to sink. Before any data flow analysis can be conducted, these source and sink methods must be identified. Listing 5.1 illustrates where the user's device ID is read and sent as SMS message. Here, the `getDeviceId()` method (called on Line 6) is the source and the `sendTextMessage` method (called on Line 9) is the sink. One of the data flow paths is represented by lines 6 → 9. Another data flow path is in lines 11 → 13, where the current time is written in the log file.

```
1  public void onStart(Intent intent, int startId)
2  {
3      /* Access sensitive data */
4      TelephonyManager tm;
5      tm = (TelephonyManager) this.getSystemService(TELEPHONY_SERVICE);
6      sensitiveData = tm.getDeviceId();
7      /* Leak the data */
8      SmsManager smsMgr = smsMgr.getDefault();
9      smsMgr.sendTextMessage("1800 8080", null, sensitiveData, null, null);
10     /* Access non-sensitive data */
11     Date currTime = Calendar.getInstance().getTime();
12     /* Write it on Log */
13     Log.d("Current time is " + currTime);
14 }
```

Listing 5.1: Simple Data Leak Example

5.1.1 Sensitive Sources and Sinks

To detect privacy leakage, we are interested in the former data flow path as the data accessed using sensitive source is leaked to potentially sensitive sink. For this we need to formally define sensitive sources and sensitive sinks.

Android categorizes some permissions as 'dangerous'. These permissions are used for accessing data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps [124]. We call such permissions as (DangerPerms). In our research work, any value, access of which requires DangerPerms, is sensitive information and any API used for accessing this is termed 'sensitive API' call. In other words, sensitive API calls (SAPICalls) denote the set of all API calls whose invocation is protected with permission from DangerPerms. For example, `getDeviceId()` is protected with dangerous permission `READ_PHONE_STATE` [125]. This set SAPICalls is partitioned into SSources and SSinks

- *Sensitive Sources*: We let SSources denote the set of all sensitive API calls that read and return a value. For example, `getDeviceId()` returns unique IMEI number of the device, and `getLatitude()` returns current latitude coordinates of the device. A sensitive source hence retrieves information that is guarded by dangerous permission and hence called sensitive information/data. Such information should not be leaked.

- *Sensitive Sinks*: We denote SSinks as the set of all sensitive API calls that send data out. For example, `sendTextMessage(message)` sends some message out of the device. A sensitive sink thus outputs information using a medium that is protected by dangerous permission. This medium presents the risk of leaking information.

The goal of the data flow analysis is to identify the connections between accessing some data (source) and transmitting that data (sink). For that, sensitive API calls (SAPICalls) are categorized into sources and sinks. Manual partitioning is not feasible as Android has more than 110,000 API methods. For this reason, we used SuSi [126], an automated machine learning based classifier that classifies all permission based Android API methods [127] into *sources*, *sinks*, and *neither* for a given Android version.

5.1.2 Taint Analysis

Android phones being pervasive attracts malicious developers to embed code in Android apps to steal sensitive data. For instance, a benign app may use our geographical location to notify us regarding some weather knowledge, while a malicious app may collect such information for the purpose of tracking an individual and stalking him/her.

In this scenario, *taint analysis* is the best approach to identify where the information has been passed. Taint analysis is the analysis based on labelling/tainting sensitive data and tracks the path of that data flow. It allows to trace if any source (geographical location) can reach an undesired sink (URL of third-party server).

Taint analysis can be done either statically [117] or dynamically [53]. In static taint analysis, the app code is analyzed, and a control flow graph of the same is constructed. This approach taints the sensitive data sources and sinks, and follow the data flow until it reaches a tainted sink from a tainted source. While dynamic taint analysis runs the

app and tries to analyze the flow of data. In this work, we leverage FlowDroid that uses static taint analysis approach.

5.2 Proposed Approach: FlowMine

Our approach considers the behaviour of an app towards sensitive information. It employs Flowdroid [38] static analysis tool to identify the data flows in a variety of apps. To classify apps, we consider the probable behavioural similarity (in terms of sensitive data usage) of an unknown sample towards benignity or malignity. Checking for similarity is done for both the behaviours, i.e., benignity and maliciousness, as one-sided similarity check may result in wrong analysis.

5.2.1 Motivating Example

To establish the fact that the behaviour of apps towards sensitive data is an important factor to differentiate benign apps from malicious, we considered *Facebook*[1] app from Google play store and *com.keji.danti604* app from Virus Share and run static taint analysis on them. Table 5.1 shows the extracted data flows. We observed that Facebook app accesses sensitive data for the synchronization of various app components. The app takes network and database information and passes them to logs or to other components via intents. On the other hand, flows in *com.keji.danti604* app directly leaks unique identifier to the web server. Thus, we can determine the behavioural difference of a benign-ware from a malware towards sensitive data.

[1]Version: 20.0.0.25.15

Facebook	
Source	**Sink**
AccountManager.get()	ContentResolver.setSyncAutomatically()
SQLiteDatabase.query()	Log.d()
Uri.getQueryParameter()	Activity.setResult()
Uri.getQueryParameter()	Log.w()
com.keji.danti604	
Source	**Sink**
TelephonyManager.getSubscriberId()	URL.openConnection()
TelephonyManager.getSubscriberId()	URL.openConnection()

TABLE 5.1: Flows in Android Facebook app and com.keji.danti604 app

5.2.2 Implementation Details

FlowMine's architecture is illustrated in Figure 5.1 and the broad steps involved in designing and implementing FlowMine are explained in the subsequent sections.

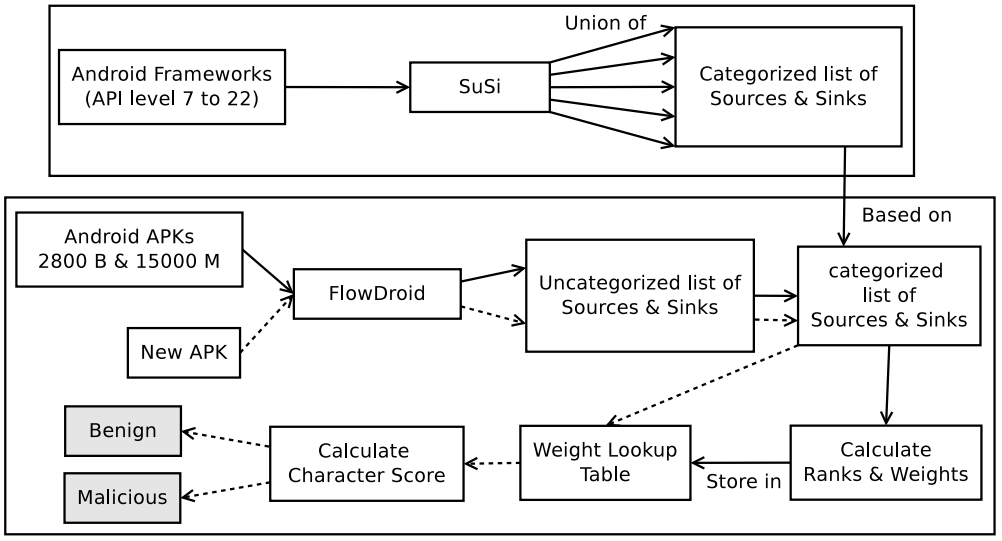


FIGURE 5.1: FlowMine Architecture

5.2.2.1 Mining Apps

FlowMine employs Flowdroid static analysis tool as shown in Figure 5.1 to identify the data flows in an app. For each app under analysis, Flowdroid (using SuSi) determines the connected sensitive data sources and sensitive data sinks. As a result, we get a set

called *flowset*, of the related source and sink pairs.

$$flowset(app) = \{source_1 \rightsquigarrow sink_1, source_2 \rightsquigarrow sink_2, ...\}$$

It is to be noted that one source may be connected to multiple sinks and many sources can sink into a single sink. But, each pair in the *flow set* is unique. We mined 2800 benign apps extracted from Google Play Store and 15000 malware samples taken from Virus Share [90] and Genome project [128]. For each app, we obtain the data flow paths, i.e., the path from sensitive source to the sensitive APIs where the data sinks. Since each app can have more than one source-sink pair in its *flowset*, we get an extremely large number of such pairs.

5.2.2.2 Flow Specificity

In general, the sources and sinks are stated with complete signature and method name. So, the data flow we determined in the previous step is studied on the basis of granularity, from finest to broadest level. Each source or sink is a method having an appropriate signature, and each method is a part of a particular class. Therefore, the specificity of the flow can be described on the following basis:

- Method: This will consider the full method signature,
  For example, `LocationManager.requestLocationUpdates(...)`
- Class: In this level, only class names are considered to express the flow
  For example, `TelephonyManager`, `LocationManager`, etc.
- Category: This is the highest abstraction level in which SuSi categories are considered for determining the flows
  For example, `LOCATION_INFORMATION`, `UNIQUE_IDENTIFIER`, etc.

We have a total of 17 abstract sources and 14 abstract sinks that makes a state space of 238 (17×14) categorized source-sink pairs. Table 5.2 specifies the flows as per category basis for the previous example of *Facebook* app discussed in Section 5.2.2.1.

Source Category	Sink Category
ACCOUNT_INFORMATION	SYNCHRONIZATION_DATA
DATABASE_INFORMATION	LOG
NETWORK_INFORMATION	INTENT
NETWORK_INFORMATION	LOG

TABLE 5.2: Flows in Android Facebook app, by SuSi categories

5.2.2.3 Assignment of Ranks and Weight

FlowMine assigns each path (SuSi source, SuSi sink pair) two ranks, viz., Benignity Rank (R_B) and Malignity Rank (R_M). These ranks are assigned on the basis of the frequency of usage in benign (malicious) apps. The popularity can be described in both benign context and malicious context considering the fact that how much any categorized source-sink pair is being used by benign apps and malicious apps respectively. On the basis of data flow analysis of large number of samples, we observed that certain source-sink pairs are more prevalent in benign group while some are prevalent in malware group. The remaining pairs can not be associated with any single class as these occur with almost same frequency in both benign/malicious apps and, as such, are not useful from classification perspective.

If a pair of source and sink is mostly used by malicious families, then Malignity Rank (R_M) of that pair is considered to be the highest. If a source-sink pair is mostly used by benign apps, then its Benignity Rank (R_B) will be highest. We have in total 238 categorized source-sink pairs. So, the ranking is done on the basis of prevalence in benign/malicious classes. It is not necessary that a source-sink pair having highest R_M rank will have lowest R_B rank or vice versa. Rank in a respective category is based on the frequency of occurrence of a pair relative to other pairs. A source-sink pair having a high occurrence in the benign group can have a high occurrence in the malicious group also. Such pairs are also not good discriminators when it comes to using this as a feature for classification. We are interested in pairs having (i) high R_B and low R_M and (ii) high R_M and low R_B. In addition, there are certain pairs which are used by both benign and malicious apps approximately in same proportions (like `DATABASE_INFORMATION -> LOG`), and there are pairs which are used by neither of the two (like `CALENDAR_INFORMATION -> BLUETOOTH`).

For assigning the *weight* to each source-sink pair, FlowMine first calculates the usage percentage of the pair in both benign samples and malicious samples. Then the weight is calculated as shown in Listing 5.2

```
1 weight = abs(% use in benignware − % use in malware);
2 if (% use in benignware < % use in malware)
3 weight = −1 ∗ weight;
```

Listing 5.2: Weight Assignment

Hence, we get a static list of weights for each source-sink pair. This list is known as *Weight Lookup Table* as shown in Table 5.3. Positive weight identifies that the corresponding pair is mostly used by benign-ware while, negative weight shows that the pair is mostly used by malware.

Categorized Source-Sink	**Weight**
DATABASE_INFO → INTENT	+0.1263961
CONTENT_RESOLVER → NETWORK	-0.0366193
CONTACT_INFO → SMS_MMS	-0.0388739
ACCOUNT_INFO → SYNC_DATA	+0.023235
...	...

Table 5.3: Sample Weight Lookup Table

5.2.2.4 Classification of an App

To classify an app as benign or malicious, FlowMine determines a *Character Score* C_a of the app. For calculating C_a, the app is analyzed by Flowdroid tool, where the *flowset* of that app is determined. Weights for each pair of source-sink present in the *flowset* are summed up. Let, the app has n source-sink pairs, the character score of the app x is computed as:

$$C_a(x) = \sum_{i=1}^{n} W_{ss}^{i}$$

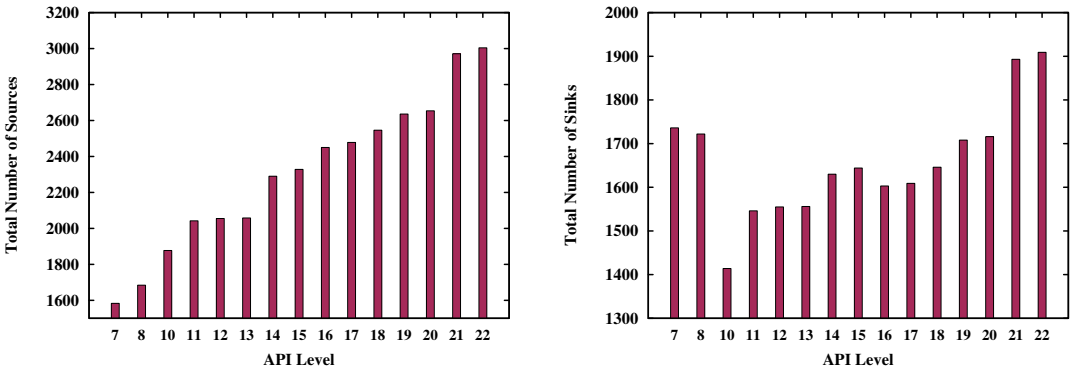
If the character score of an app comes out to be positive, then FlowMine classifies input app as a benign app as its behavior towards sensitive data is more similar to benign-ware than malware. While, if the character score comes out to be negative, then it is classified as malicious app.

5.3 Experimental Evaluation

We evaluated FlowMine on DroidAnalyst server [129]. Taking into consideration the fact that static taint analysis technique consumes a lot of memory, we implemented FlowMine with a simple ranking computation technique based on probability distribution.

5.3.1 Dataset Preparation

To identify the exhaustive list of sources and sinks, we analyzed all Android APIs from version 7 to version 22. Figure 5.2(a) shows the number of sources corresponds to each API level and Figure 5.2(b) shows the number of sinks. We took the union of the extracted sources and sinks from all the analyzed Android APIs. That list we provide as an input to our taint analysis tool. We extracted 2800 benign apps from Google Play Store and 15000 malicious samples from Virus Share and Genome project.



(a) Total Number of Sources (b) Total Number of Sinks

FIGURE 5.2: Total Number of Sources and Sinks

5.3.2 Analysis Results

FlowMine identifies the sensitive data flow paths in both malware samples and benign samples. The difference found in the presence of these paths acts as a measure to determine the degree of similarity of any unknown test app towards either of the two.

5.3.3 Data Flow in Benign Apps

We examined 2800 benign apps and analyzed the flows. Since each app has a *flowset* associated with it and the set contains one or more source-sink pairs, we have a total of 338619 pairs of sources and sinks. To reduce the complexity, we abstract sources and sinks as per SuSi categorization. We also computed the percentage of each pair being used across available benign and malicious apps in our dataset. Table 5.4 summarizes data flows in our benign data set samples.

SOURCE	LOG	INTENT	NETWORK	FILE	SYS_SET	ACC_SET	SMS_MMS	AUDIO	NFC	SYNC_DATA	CALE_INFO	LOC_INFO
DATABASE_INFO	1.9618	1.8233	0.0375	0.0289	0.0318	0.0126	0.000	0.0180	0	0	0	0.0008
CALENDAR_INFO	1.0422	2.1260	0.0720	0.0900	0.0558	0.0050	0	0.0035	0.0044	0	0.0218	0.0008
NETWORK_INFO	1.4098	0.4515	0.0360	0.0206	0.0156	0.0109	0.0251	0.0070	0	0	0	0
LOCATION_INFO	1.1098	0.3667	0.0109	0.0064	0.0035	0	0	0.0044	0.0005	0	0	0.0083
CONTENT_RES	0.5224	0.2430	0.0218	0.0070	0.0064	0.0005	0.0014	0.0035	0.0011	0	0.0005	0.0002
UNIQUE_ID	0.2350	0.1517	0.0050	0.0002	0.0005	0.0008	0.0002	0.0014	0	0	0	0
ACCOUNT_INFO	0.1408	0.0779	0.0053	0.0005	0.0017	0.0785	0	0.0005	0	0.0236	0	0
CONTACT_INFO	0.1151	0.0271	0	0	0	0.0014	0	0	0	0	0	0
FILE_INFO	0.0738	0.0401	0.0011	0.0041	0.0014	0	0	0.0005	0	0	0	0
NFC	0.0156	0.0380	0.0020	0.0017	0	0	0	0	0.0215	0	0	0
BLUETOOTH_INFO	0.0451	0.0067	0	0	0.0002	0	0	0.0005	0	0	0	0
SMS_MMS	0.0124	0	0	0	0	0	0.0239	0	0	0	0	0
SYNCH_DATA	0.0056	0	0	0	0	0	0	0	0	0.0029	0	0
IMAGE	0.0044	0	0	0	0	0	0	0	0	0	0	0
BROWSER_INFO	0.0014	0.0008	0	0	0	0	0	0	0	0	0	0
SYSTEM_SET	0.0014	0	0	0	0	0	0	0	0	0	0	0

TABLE 5.4: Categorized data flows (in %) for benign apps

We infer that most important sources are `DATABASE_INFORMATION`, `CALENDAR_INFORMATION`, `NETWORK_INFORMATION`. This signifies that most of the Android apps interact with the outer environment by extracting information from the databases. The frequently used sinks are `LOGS` and `INTENT` that constitute about 69% of total sinks for all sources. This is due to the fact that logs and intents are highly used for component interactions. `LOG` is a sink but is less harmful in newer Android versions as these logs can only be accessed with administrative privileges. The least frequently used sources are `BROWSER`, `IMAGE`, `SYSTEM_SETTING` and they rarely end up in sensitive sinks.

5.3.4 Data Flow in Malicious Apps

As per our analysis, we obtained 1493027 distinct source-sink pairs while analyzing 15000 malicious samples. We used SuSi categories to reduce the complexity and determine the percentage of source-sink pairs being used. Table 5.5 summarizes the data flows in our malicious data-set samples.

SOURCE	LOG	INTENT	NETWORK	FILE	SYS_SET	ACC_SET	SMS_MMS	AUDIO	NFC	SYNC_DATA	CALE_INFO	LOC_INFO
DATABASE_INFO	1.4054	1.6969	0.0268	0.0222	0.0021	0.0002	0.0072	0.0141	0	0	0	0
CALENDAR_INFO	0.3779	0.2407	0.0062	0.0049	0.0040	0	0.0002	0.0044	0	0	0	0
NETWORK_INFO	2.0025	0.2203	0.3461	0.0249	0.0048	0.0002	0.7896	0.0077	0	0	0	0
LOCATION_INFO	1.5086	0.1225	0.0830	0.0004	0	0	0.0045	0.0009	0	0	0	0.0004
CONTENT_RES	0.6621	0.2215	0.0574	0.0052	0.0019	0	0.0024	0.0059	0	0	0	0
UNIQUE_ID	0.7347	0.0788	0.4421	0.0160	0	0	0.0388	0.0011	0	0	0	0
ACCOUNT_INFO	0.0057	0.0076	0.0003	0	0.0001	0.0058	0	0.0001	0	0.0004	0	0
CONTACT_INFO	0.0275	0.0184	0	0	0.0007	0	0.0388	0.0008	0	0	0	0
FILE_INFO	0.0472	0.0170	0	0	0.00080288	0	0	0.0021	0	0	0	0
NFC	0.0024	0.0004	0.0001	0	0	0	0.0002	0	0.0007	0	0	0
BLUETOOTH_INFO	0.0052	0.0148	0.0006	0.0004	0	0	0	0.0014	0	0	0	0
SMS_MMS	0.0022	0.0012	0	0	0	0	0.0078	0	0	0	0	0
SYNC_DATA	0.0002	0	0	0	0	0	0	0	0	0	0	0
IMAGE	0	0	0	0	0	0	0	0	0	0	0	0
BROWSER_INFO	0.0010	0.0008	0	0	0	0	0	0	0	0	0	
SYSTEM_SET	0	0.0002	0	0	0	0	0	0	0	0	0	0

TABLE 5.5: Categorized data flows (in %) for malicious apps

From the table, we observed that most prominent source is `NETWORK_INFORMATION`, as it is used almost two times than the usage of `DATABASE_INFORMATION`. Also, the use of `NETWORK_INFORMATION` is just double than its use in benign samples. The `CALENDER_INFORMATION` is accessed with just half the rate at which it is being used in benign samples. The most astonishing sink used by malware is `SMS_MMS` in comparison to its use in benign samples. Most of the sensitive sources leak into `SMS_MMS` sink[2]. Out of total app we analyzed, 22% of them use SMS as the sink to send sensitive source data (like `UNIQUE_ID`, `LOCATION_INFO`, `DATABASE_INFO`, etc.) and hence causing privacy leakage.

[2]Premium SMS text message frauds are prevalent during this time

5.3.5 Accuracy

Our test dataset consists of both benign and malware samples. If we consider the over classification, FlowMine classifies an unknown sample with an accuracy of 96.5%. A 10-fold evaluation of FlowMine shows that it can correctly classify 96% of all benign apps and 97.2% of novel malware samples. The higher detection rate for malware samples is due to the fact that malware is self similar in nature. Different malware constitutes the same attack again and again and may share similar code. Thus, its detection is higher in our proposed method.

FlowMine classifies 97.2% of malware with a false positive rate of 3.5%. It determines malware is leaking sensitive information with an accuracy of 98%. It is to be noted that while evaluating a single app, each pair of *flowset* needs to be considered. Considering only the most sensitive source/sink leads to higher false positive rate as malware can be repackaged with same source ending on different sinks and vice versa.

5.4 Summary and Limitations

Static data flow analysis is a means for automatically enumerating the data flows inside a program. FlowMine considers the data flow path from a sensitive source to a data sink. These paths are further ranked by assigning weights, which is the absolute difference between its use in benign and malicious samples. The higher difference means that path plays a vital role in classifying apps. Our proposed approach is faster as the ranking mechanism is simpler. It can do even better if, the rankings and weights assignment mechanism can be more precise. So, considering more benign and malware samples during analysis will help us to give more accurate results.

FlowMine focuses on the detection of the leakage path in a single app. But, an attacker can pass sensitive information within a wrapper of a path that scattered across multiple apps. An app can even bypass Android security model by exploiting transitive permission usage to escalate their privileges and leak sensitive information. Single app analysis techniques are no longer able to detect such leakage paths. So, our next work will focus on analyzing multiple apps together to detect privacy leakage through multiple apps.

Part II

Inter App Analysis

Chapter 6

ICC Primitives based Static Analysis

As discussed in the previous chapters, single app analysis techniques dominate security assessment in Android to detect malware. Although, an app can bypass Android security model and leak sensitive data by colluding with other app [130]. We shall be discussing collusion in detail in this chapter and proposing an approach to address such leakage.

6.1 Introduction

Android attracts a large number of users because of ease of using apps and enhanced functionalities. In order to provide these, an app needs to access end user's personal information. For example, to locate nearby restaurants, the app needs to access user's location information, to share pictures, the app needs to access user's photo gallery. To develop rich apps, developers can leverage data and services provided by other apps. For example, a cab booking app can ask Google Maps for client's or driver's location information. This communication between apps can reduce developer's burden and facilitate functionality reuse. To provide such communication Android framework offers message passing mechanism called Intents. Android app does not require any special permission for communication. Also, Android framework is not designed to protect the information that is going outside an app.

Inherent sharing of data in such communication exposes app(s) vulnerabilities like data leak, confidential user data breach, unauthorized access, privilege escalation, etc. Malware writers may exploit permission model imposed by Android through Intent based Inter-Component Communication (ICC). This is also known as *Privilege escalation attack* at app level [131] or *Collusion attacks* [132], [133]. Collusion refers to the scenario where two or more apps with a limited set of permissions communicate with each other to gain indirect privilege escalation and can perform unauthorized actions. Android's security framework is not sufficient for transitive policy enforcement allowing privilege escalation attacks as shown by many examples [7, 134–136].

6.1.1 Threat Model

To demonstrate collusion attack, we developed a motivation example as shown in Figure 6.1. In this example, there are two apps. The first app named *msgRead* needs to ac-

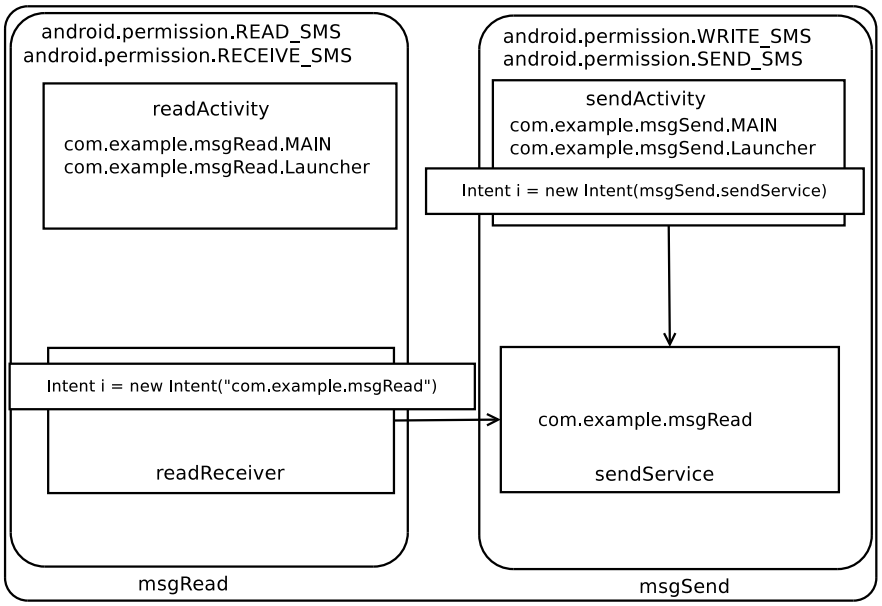


FIGURE 6.1: msgRead app is colluding with msgSend app leads to privilege escalation

cess users' private data, i.e., SMS in this case. For that, the app developer mentions two permissions `android.permission.READ_SMS` and `android.permission.RECEIVE_SMS` in `AndroidManifest.xml` of this app. Similarly, the second app named *msgsSend* needs to write and send permission, and they are mentioned in its `AndroidManifest.xml` as `android.permission.WRITE_SMS` and `android.permission.SEND_SMS`. These two apps

can communicate with each other through intents. Individually both the apps have limited set of permissions as *msgRead* can read the SMS but cannot send it to the outside world, and *msgSend* can send SMS but it cannot read any SMS.

In the attack scenario *msgRead* can read the text message and pass it as an intent parameter. This intent is received by *msgSend* app, that can send the message to the outside world. Although *msgRead* has no permission to send SMS, it can do that transitively and hence escalate its privileges. *msgSend* app can be a genuine app or malicious app. If it is genuine, it gets exploited by the *msgRead* app.

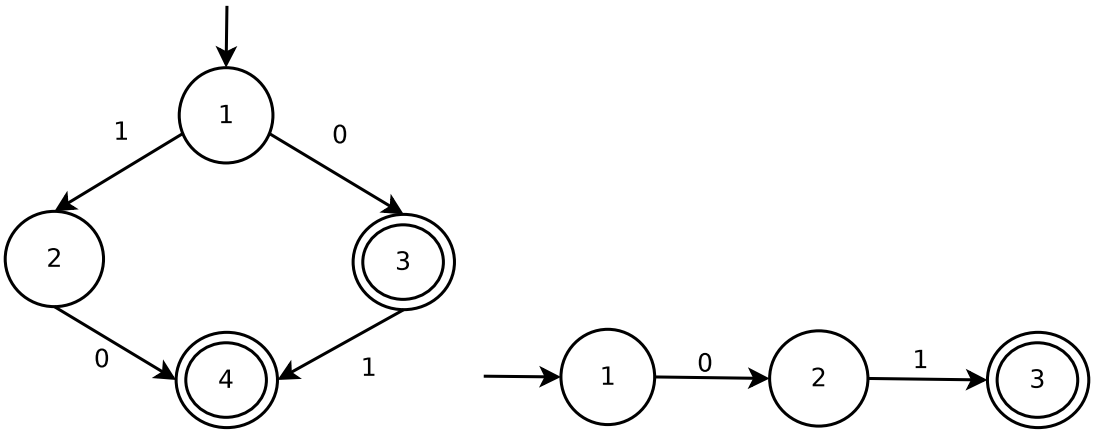
6.1.2 Automaton Model

As Android does not enforce any special permission for app communication, evaluating the possibilities of attack due to inter-app communication becomes increasingly more important. Our approach of ICC collusion detection is based on an automaton model. The problem of collusion detection can be visualized as string searching and pattern matching problem as any mismatch of permission between sender and receiver components of an intent may lead to privilege escalation. Such problems can be solved by finite automaton. To the best of our knowledge, this is the first work where the automata approach has been used for collusion detection in Android.

A *finite state automaton*, denoted $\mathcal{M}$, is a $5-$tuple $\langle S, \Sigma, \delta, s_0, F\rangle$ where S is a set of finite states, Σ is the automaton alphabet (a set of finite symbols), $\delta : S \times \Sigma \rightarrow S$ is the *transition function*, $s_0 \in S$ is *start state* and, $F \subseteq S$ is the set of *final states*. The *size* of automaton $\mathcal{M}$ is simply the number of its states. Let w be a word on the alphabet Σ, that is $w = a_1, a_2, \cdots a_k$ and $w \in \Sigma^*$. The word w is acceptable by automaton $\mathcal{M}$ iff for w, there exists a sequence of state $s_1, s_2, \cdots, s_k$ such that $s_1 = \delta(s_0, a_1)$ and $s_k \in F$, for all $i \in \{2, \cdots, k\}$, $s_i = \delta(s_{i-1}, a_i)$. The set of such words forms a language $L(\mathcal{M})$ that is *recognizable* by $\mathcal{M}$. Example automata $\mathcal{M}$ is illustrated by Figure 6.2(a) where $\Sigma = \{0, 1\}$ and $L(\mathcal{M}) = \{0, 01, 10\}$.

6.1.3 Intersection Automaton

Given two automata $\mathcal{M} = \langle S, \Sigma, \delta, s_0, F\rangle$ and $\mathcal{M}' = \langle S', \Sigma', \delta', s'_0, F'\rangle$, *intersection automaton*, $\mathcal{I}$, of $\mathcal{M}$ and $\mathcal{M}'$, denoted $\mathcal{M} \cap \mathcal{M}'$, is defined as follows:



(a) Automata $\mathcal{M}$ that accepts $\{0, 01, 10\}$ (b) Automata $\mathcal{M}'$ that accepts $\{01\}$

FIGURE 6.2: Total Number of Sources and Sinks

$\mathcal{I} = \langle S_I, \Sigma_I, \delta_I, \{s_0, s_0'\}, F_I \rangle$, where

$S_I = \{(s, s') \mid s \in S \text{ and } s' \in S'\}$,

$\Sigma_I = \Sigma \cup \Sigma'$,

$F_I = \{(s, s') \mid s \in F \text{ and } s' \in F'\}$, and

$\delta_I : S_I \times \Sigma_I \to S_I$, with for all

$((s_1, s_1'), (s_2, s_2')) \in S_I \times S_I$, and $a \in \Sigma_I$, $\delta_I((s_1, s_1'), a) = (s_2, s_2')$ iff $\delta(s_1, a) = s_2$ and $\delta'(s_1', a) = s_2'$.

The intersection of automaton $\mathcal{M}$ and $\mathcal{M}'$ of Figures 6.2(a) and 6.2(b) respectively is illustrated in Figure 6.3. Let $\mathcal{M}_1, \mathcal{M}_2, \cdots, \mathcal{M}_m$, be a collection of m finite state

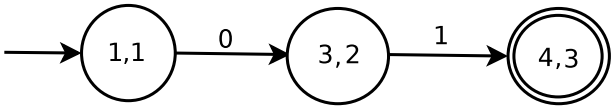


FIGURE 6.3: Intersection of Automata $\mathcal{M}$ and $\mathcal{M}'$ that accepts $\{01\}$

automata of sizes $n_1, n_2, \cdots, n_m$ respectively. One interesting problem in automaton theory is checking whether the intersection of their recognizable languages is empty, i.e., decide on the following question: $\cap_{i=1}^{m} L(\mathcal{M}_i) = \emptyset$? Let (Q) denote this question. The standard algorithm for answering the above question involves constructing the finite state automaton corresponding to the intersection $\mathcal{I} = \mathcal{M}_1 \cap \mathcal{M}_2 \cap \cdots \cap \mathcal{M}_m$, and solving the emptiness problem for $\mathcal{I}$: $L(\mathcal{I}) = \emptyset$. The size of the intersection automaton $\mathcal{I}$ is $O(n_1 \times n_2 \times \cdots \times n_m)$. The last observation on the size of intersection automaton shows that using such construction is memory consuming. We know from [137] that there is no more efficient solution to answer question (Q).

6.2 Proposed Approach

In this section, we start with the explanation of preliminaries of our proposed approach followed by detailing of each step and finally applying the approach on the motivating example shown in Figure 6.1. A typical process to detect collusion proceeds by representing an individual app as a graph (application automaton). Application graphs of two apps A and B are connected by an edge if any component of A is communicating with any component of B via Intent. Such graphs are called *application automaton*. To detect collusion we need to check that these edges do not violate permission model for which we define some collusion policies and represent these policies as *policy automaton*. Collusion detection is performed through the intersection of application automaton and policy automaton.

6.2.1 Application Automaton

First, we formalize *application automaton*, the data structure used to represent all possible Intent based ICC interactions between apps.

Definition 6.1 (Application Automaton)**.** An *application automaton*, or Λ, is a tuple $G = \langle S, \Sigma, \delta, s_0, F \rangle$ where S is the set of all the components present in all the apps under analysis and one additional state s_0, Σ is the set of permissions required by each app under analysis, $\delta : S \times \Sigma \rightarrow S$ is the *transition function* which takes permission needed to move to the component. There is also a transition from s_0 to all the other states in $S \setminus \{s_0\}$. All the states, except s_0, are considered as final. That is $F = S \setminus \{s_0\}$.

6.2.1.1 Constructing Application Automaton

Constructing app automaton is a crucial part of performing collusion detection. It covers all possible intent based ICC interactions between apps. Following are the steps to construct application automaton:

(a) **Represent app as graph**: In this step, we are capturing all the action strings associated with components of the app and all ICC calls (both implicit and explicit). Let A be an app. We first define the following sets:

α, β, and γ are the set of all the *activities*, *services*, and *broadcast receivers* present in A respectively. ξ is the set of *intent API calls* present in A , and $\varsigma(\xi)$ is the set of all the *action strings* passed in ξ.

Therefore, we can define the set of vertices V as the set

$$V = \alpha \cup \beta \cup \gamma \cup \varsigma(\xi) \tag{6.1}$$

We also define the following two sets:
$S(\xi)$ is the set of vertices which are sources of intents from ξ, that is

$$S(\xi) = \{x \in \{\alpha \cup \beta \cup \gamma\} \mid x \text{ initiates an intent } i \in \xi\},$$

$T(\xi)$ is the set of vertices which are targets of intents from ξ, that is

$$T(\xi) = \{x \in \{\alpha \cup \beta \cup \gamma\} \mid x\text{receives an intent } i \in \xi\}.$$

Then, the set of edges E is defined as follows:

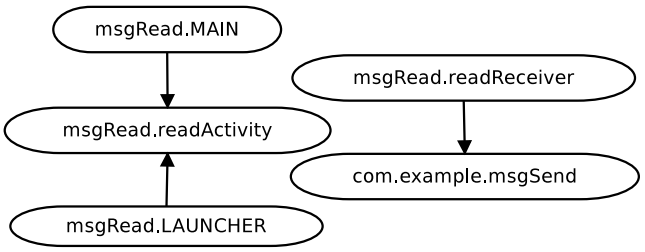
$$\begin{aligned} E = \{(x,y) \mid x \in S(\xi) \text{ and } y \in T(\xi)\} \cup \\ \{(x,y) \mid x \in S(\xi) \text{ and } y \in \varsigma(\xi)\} \cup \\ \{(x,y) \mid x \in T(\xi) \text{ and } y \in \varsigma(\xi)\} \end{aligned} \tag{6.2}$$

So, an app A can be represented as a directed graph $G(V, E)$ where V and E are defined by Equations 6.1 and 6.2 respectively. Visually apps `msgRead` and `msgSend` can be represented as Figures 6.4(a) and 6.4(b) respectively.

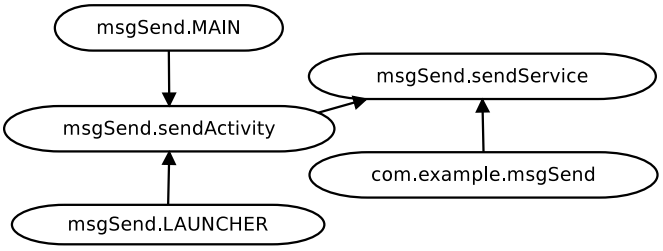
(b) **Union of the graphs**: In order to resolve inter-app implicit intent calls, in this step, we perform union operation on all the apps' graph which are under analysis. Say, there are n apps under analysis, and each of them is represented as graph by step *(a)*. The union graph G^u can be defined as:

$$G^u = \bigcup_{i=1}^{n} G_i, \quad G_i = (V_i, E_i) \quad \forall i = 1 \cdots n$$

That is, it is the graph $G^u = (V, E)$ with $V = V_1 \cup V_2 \cup \cdots \cup V_n$ and $E = E_1 \cup E_2 \cup \cdots \cup E_n \cup E'$, where E' is the set of edges that encodes inter-app implicit



(a) msgRead app



(b) msgSend app

FIGURE 6.4: Graph representation of apps

intent calls. More formally, for any $i \neq j \in \{1, \cdots, n\}$, let A_i and A_j be two apps and G_i and G_j the corresponding graphs. Let ξ_i (resp. ξ_j) be the set of intent API calls present in A_i (resp. A_j) and for $k = i, j$, let $S(\xi_k)$ (resp. $T(\xi_k)$) denote the vertices which are sources (resp. targets) of intents from ξ_k. Then we define E' as follows:

$$E' = \{(x, y) \mid \exists i, j \in \{1, \cdots, n\}$$
$$\text{s.t. } x \in S(\xi_i) \text{ and } y \in T(\xi_j) \text{ and } i \neq j\}$$

Union of graphs in Figure 6.4 are depicted in Figure 6.5.

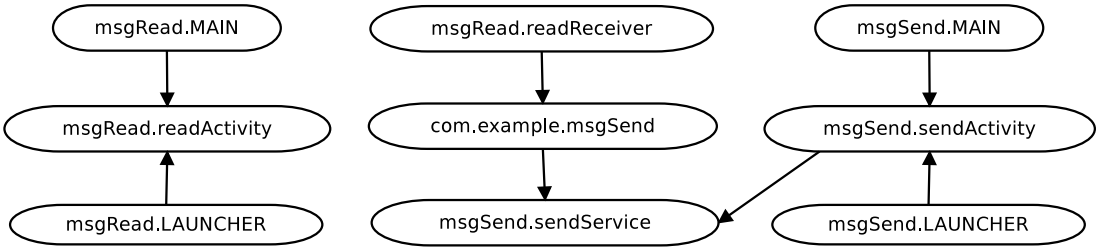


FIGURE 6.5: Union of msgRead and msgSend apps' graphs

(c) **Prune the union graph**: Now, the role of action strings (set ς) is complete, therefore in this step we prune the graph G^u by removing all action string nodes. Pruning the graph will help in improving the computational speed in further analysis. The rules to remove such nodes are: `(a)` If there is no incoming edge, then

just remove the node. (b) If there are incoming edges from components, then connect incoming edge component to outgoing edge component directly and remove the action string node. The pruned graph $G^p(V^p, E^p)$ can be defined as follows:

$$E^p = E \cup \{S(\xi) \to T(\xi) \mid S(\xi) \to x \to T(\xi), x \in \varsigma\}$$
$$\setminus\{\{(x, y) \mid x \in S(\xi) \text{ and } y \in \varsigma(\xi)\} \cup$$
$$\{(x, y) \mid x \in T(\xi) \text{ and } y \in \varsigma(\xi)\}\}$$

$$V^p = V - \varsigma(\xi)$$

Pruned graph of Figure 6.5 is illustrated in Figure 6.6.

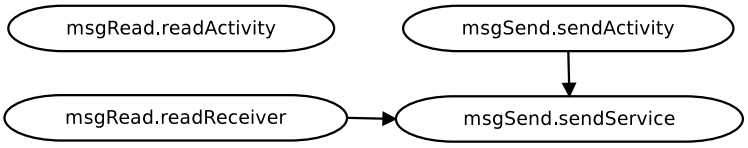


FIGURE 6.6: Pruning of union graph in Figure 6.5

(d) **Convert pruned graph in automaton**: Finally, graph G^p is converted to an automaton Λ. All the nodes of G^p form the set F in Λ. Include one additional state, that becomes s_0 in Λ. $S = F \cup \{s_0\}$ in Λ. δ includes transitions from $s_0 \to F$ with permissions required to access element of F and all the edges of G^p. Application automaton of example apps is illustrated in Figure 6.7.

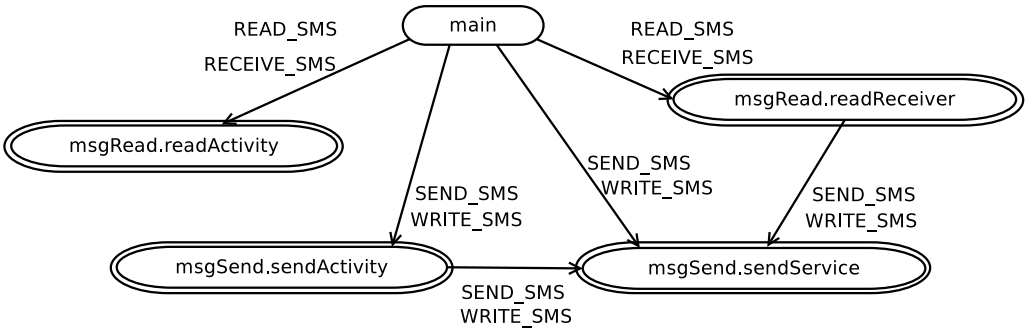


FIGURE 6.7: Application Automaton

6.2.2 Policy Automaton

Here, we define formally *policy automaton*, and the data structure used to represent collusion policies.

Definition 6.2 (Policy Automaton)**.** A *policy automaton* or Γ is a tuple $G = \langle S, \Sigma, \delta, s_0, F \rangle$ where S is the set of all states in Γ (we will explain it later), Σ is the set of *dangerous permissions* (cf. Section 2.2.3), $\delta : S \times \Sigma \rightarrow S$ is the *transition function* which takes $x \mid x \in \Sigma$ and move to $y \mid y \in S$, $s_0 \in S$ is the first state of policy automaton, and, F is the set of all states that led to collusion.

6.2.2.1 Constructing Policy Automaton

Constructing policy automaton defines the precision and accuracy of detection. It depicts inter-app order of permissions which may lead to collusion. To construct policy automaton, first we need to define colluding rules. Collusion is said to have occurred if dangerous permissions place in certain order within two or more apps. In our approach we assume that if the first app has been granted some dangerous permission and is communicating with the app having permission to send data, as a result of ICC, dangerous permissions are granted but not requested by the second app. To send data to the outside world, an app requires any of the three permissions: `WRITE_EXTERNAL_STORAGE`, `SEND_SMS` and `INTERNET`.

If dangerous permission is followed by the above permissions or ϵ, where ϵ refers to a null set of permission, there exists collusion. The sample policies are illustrated in Listing 6.1.

```
1 READ_PHONE_STATE → INTERNET
2 ACCESS_FINE_LOCATION →    SEND_SMS
3 READ_CONTACT → WRITE_EXTERNAL_STORAGE
4 ACCESS_NETWORK_STATE → ϵ
```

Listing 6.1: Sample Rules

Finally, as mentioned in Algorithm 6.1, these policies are converted into automaton Γ. An example is shown in Figure 6.8.

Algorithm 6.1 Construct Policy Automaton

PolicyAutomata()

Define colluding policies ;

Γ = Policies2Automaton(); // Convert Policies to Automaton as explained in 6.2.2.1

return (Γ);

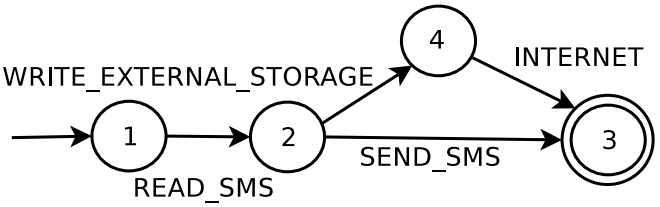


FIGURE 6.8: Policy Automaton

6.2.3 Collusion Detection

As illustrated in Algorithm 6.2, we make use of application automaton Λ and policy automaton Γ to detect collusion.

Algorithm 6.2 Collusion Detection through Intersection Automaton

CollusionDetection(APPs)

Input: APPs is the set of all APKs under test

$i = 0$;

$n = |APKs|$; // Size of APKSet

repeat

$A_i = i^{th}$ APK of APPs;

G_i = App2Graph(A_i); // Convert APK to Graph

$G^u = G^u \cup G_i$;

until ($i <= n$)

G^p = Prune(G^u); // Remove all action strings

Λ = Graph2Automaton(G^p); // Convert Graph to Automaton

Γ = PolicyAutomata();

$\Psi = \Lambda \cap \Gamma$;

if Ψ contains final states **then**

return (TRUE);

else

return (FALSE);

In Section 6.2.1, we built automaton machine that accepts the communication among apps and in Section 6.2.2, we built a machine that only accepts colluding permission sequence. Now, the intersection of the two will tell whether colluding permission sequence exists in the communication of apps or not. If the intersection has non-empty final states, it means that the apps are colluding with each other as depicted in Figure 6.9.

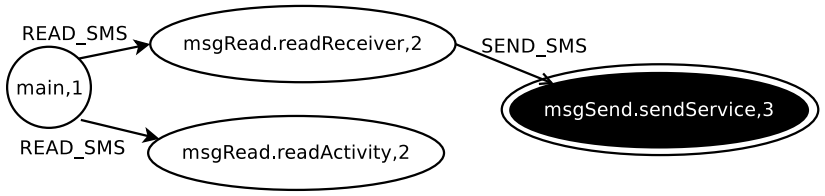


FIGURE 6.9: Collusion Detection through Λ and Γ

6.3 Evaluation

In this section, we evaluate results from our experiments to judge the efficacy of our proposed tool. During all the experiments we have used a PC with an Intel Core i3 2.0GHz CPU processor with 4GB RAM.

6.3.1 Dataset Preparation

Due to lack of availability of benchmark apps that exhibits collusion, we created our own dataset of fourteen test examples that depict the real-time privilege escalation collusion attacks. Out of fourteen, one is inter-device communication, others collude through services and broadcast receivers. We named the dataset as MNIT dataset. We also took three colluding samples from DroidBench [138] and four apps from Google Play Store [88].

6.3.2 Analysis Results

In this section, we present results obtained by executing the proposed approach on Droidbench Inter-App Communication dataset [138], some set of real-world apps from Google Play Store [88], and MNIT dataset.

DroidBench: DroidBench test suite has thirteen categories, out of which one is inter-app communication which consists of three apps. Our approach has successfully detected two cases of collusion with the path of information leakage and one case of no-collusion. Echoer app is colluding with the other two apps called sendSMS and startActivity-ForResult1. It does not have any permission yet it possesses device ID and location information. The results are shown in Table 6.1. (✓) denotes the presence of collusion, (✗) denotes the absence of collusion.

	Echoer	sendSMS	strtActForRslt1
Echoer	-	✓	✓
sendSMS	✓	-	✗
strtActForRslt1	✓	✗	-

TABLE 6.1: DroidBench Inter-App Communication dataset

Google Play Apps: We also test our approach against real apps from Google Play Store, which is considered to be benign and have download count in billions. We pick four such apps from different categories, social networking (Facebook), Messaging (Instagram), Shopping (Amazon) and Movie(IMDB). Our approach can detect one case of collusion and two cases of no-collusion out of five. We cannot detect the rest three cases because Amazon showed incompatible issues while de-compression. The results are shown in Table 6.2. (-NA-) denotes non-availability of the results.

	Facebook	Instagram	Amazon	IMDB
Facebook	-	✓	-NA-	✗
Instagram	✓	-	-NA-	✗
Amazon	-NA-	-NA-	-	-NA-
IMDB	✗	✗	-NA-	-

TABLE 6.2: Google Play apps

MNIT Dataset: We tested the approach in 210 test cases illustrated by 14 apps. Our approach successfully detects eight cases of collusion and 202 cases of no-collusion. Results are shown in Table 6.3.

	msgSend	msgReceive	cam6	serverSocket	getDev	getLoc	getImage	getFData	collector
msgSend	-	✓	✗	✗	✗	✗	✗	✗	✗
msgReceive	✓	-	✗	✗	✗	✗	✗	✗	✗
cam6	✗	✗	-	✓	✗	✗	✗	✗	✗
serverSocket	✗	✗	✓	-	✗	✗	✗	✗	✗
getDev	✗	✗	✗	✗	-	✗	✗	✗	✓
getLoc	✗	✗	✗	✗	✗	-	✗	✗	✓
getImage	✗	✗	✗	✗	✗	✗	-	✗	✓
getFData	✗	✗	✗	✗	✗	✗	✗	-	✓
collector	✗	✗	✗	✗	✓	✓	✓	✓	-

TABLE 6.3: MNIT dataset

6.3.3 Timing Analysis

Relevant information extraction from an app and representing that into an automaton model is time-consuming but one time process. The average size of an app is ∼700 KB and it took around 20 sec to build the model. On the other hand, intersection of application automaton with policy automaton to detect presence/absence of collusion between different apps need to be done at max n^2 times where n represents the number of apps in a dataset. But the time taken by this step is negligible as compared to the time taken by the first step. The maximum time taken to check Facebook and Instagram app is 1.8 sec. We observed that on an average, there are around 20-26 user installed apps in a device. Of these, many apps are common across many users and once their application automata have been constructed, these need not be analyzed again. Use of policy automaton allows us to enforce different policies at different levels by modifying policy automaton accordingly. Our approach is scalable in terms of number of apps as well as flexible in terms of security policy framework. Therefore, once we prepare automaton model of top 100 apps, we need to incur a low overhead for each new user. We can state that we can conclude that the proposed approach can perform reasonably well in real time.

6.3.4 Scalability

The proposed method constructs automaton by considering information at a granularity of component-level which means that each component represents one state in the automaton in contrast with app-level analysis [76] or method-level analysis [26, 71]. This reduces state space and our automaton are compact and can be processed faster. App-level analysis suffers from low precision (false positives), and method-level analysis is not scalable to many apps. The size of application automaton is order of n, where n is the total number of components in the apps under analysis. The policy automaton also has finite number of states. Detecting the collusion is then done by intersection of both application and policy automaton. This is linear and hence has $O(n)$ memory consumption.

The proposed approach shows its scalability against large *apk*s. Facebook app is the largest *apk* with respect to number of components among all the *apk*s in datasets. It has 391 components while checking collusion of Facebook with Instagram which has 53 components, total no. of states in the app automaton are 444 (391 + 53). This shows that the no. of states in application automata is linear with respect to the total no. of components in the apps under analysis. Thus we infer that the proposed approach is scalable.

6.4 Summary and Limitations

Android app collusion poses a significant threat to user privacy. We developed a novel collusion analysis model based framework that can detect ICC collusion. The model statically analyzes different apps and retrieves inter-app communication, followed by representing that communication in the form of a state machine to detect collusion. To attain this objective, we developed a state machine for colluding policies. Our experimental results demonstrate that colluding policies can be used as the pattern that can be matched against the communication of apps to detect collusion.

Currently, we consider intent based ICC communication between two apps. This work is extended in SniffDroid [139] that capture communication through shared preferences

and content providers. The involvement of more than two apps is also possible in collusion. Our next work will focus on large-scale analysis of apps using formal verification methods.

Chapter 7

Collusion Detection by Formal Model

In the previous chapter, we discussed an approach to detect privacy leakage paths planted between two apps through intent based ICC mechanism. However, construction of leakage path may involve more than two apps. But, the search space posed by possible combinations of these apps is exponential. Therefore, there is a need to imply effective methods to narrow down the search for collusion candidates of interest while ensuring that no leakage path is missed.

In this chapter, we propose a novel approach, called *SneakLeak+*, to detect potentially colluding apps. The tool extends a chain of existing tools to first extract Java bytecodes from the *apk* files, then it extracts ICC based communication channels (intents, content providers, and shared preferences) followed by identification of methods that access sensitive data. All the extracted information is utilized to eliminate the non-communicating apps and concentrate only on potential candidates for collusion in the given set of apps under consideration. To ensure scalability, the stored information is then processed to generate an abstract extended ICC model in a formal specification language. Use of formal methods in any scenario requires a precise characterization and representation of the relevant properties that need to be verified. We precisely express colluding condition(s) and app model that could not be specified by existing formalism.

7.1 Formalization

This section formally presents how inter-app leakage path can be termed as app collusion. Also, some background on formal verification and model checking is explained.

- *Dangerous Permissions:* As discussed in Section 5.1.1, we denote DangerPerms as the set of over 130 pre-defined permissions provided by the Android platform [12].
- *Sensitive API Calls:* Let SAPICalls denote the set of all API calls whose invocation is protected with a permission from DangerPerms (cf. Section 5.1.1). The set SAPICalls is partitioned into SSources and SSinks:
  - *Sensitive Sources:* We let SSources denote the set of all sensitive API calls that read and return a value (cf. Section 5.1.1).
  - *Sensitive Sinks:* We denote SSinks as the set of all sensitive API calls that sends data out (cf. Section 5.1.1).
- *Inter-Component Communication Methods:* ICCom denotes the set of API methods that are used for inter-component (IC) communications. For instance, `startActivity(), bindService(), sendBroadcast()` are some of the IC communication methods.
- *Resource-Permission Map:* It is a surjective function γ : SAPICalls $\rightarrow$ DangerPerms that maps any sensitive API call to the corresponding access permission. In the sequel, we consider a fixed resource-permission map γ developed using PSCout [140].
- *Sensitive Information:* Any value retrieved from a sensitive API call from SSources.

7.1.1 Android App Collusion

Sensitive information is retrieved, manipulated and exchanged by apps. Our goal is to detect situations where sensitive information is leaked due to interactions between several apps, which we call collusion.

Definition 7.1 (Collusion)**.** A set $\{A_1, \ldots, A_n\}$ of Android apps *collude* if they interact to allow one of them, e.g. A_i, access and output sensitive data that is protected by

permissions that have not been granted to A_i. This is a *minimal colluding set* if no proper subset of $\{A_1, \ldots, A_n\}$ collude.

Our goal is to detect collusion. We focus on the situation where the information leaked is obtained through inter-app communications. We build an abstract model of the apps and their interactions to detect collusion. Our model is built from the control flow graphs of the apps. We start by defining collusion in terms of the path in the control flow graph of apps.

Following section 2.1, an Android app A consists of a set of components $\mathsf{Comp}(A) = \{C_1, \ldots, C_m\}$, and a set of sensitive resource access permissions: $\mathsf{Perm}(A) \subseteq \mathsf{DangerPerms}$. Each component C_i is a program. For each program, we extract its control flow graph $\mathsf{CFG}(C_i)$[141]. A control flow graph (CFG) of C_i is a finite directed graph, where each vertex contains a statement[1] from C_i, and the edges in the graph represent jumps from a statement to the next statement. The control flow graph represents possible execution paths. While some of the paths may not be feasible when the run-time value(s) of variables are taken into account, every execution of the program corresponds to a path in its control flow graph. We denote $\mathsf{CFG}(A)$ the control flow graph of A which consists of the disjoint union of the control flow graphs of its components $\biguplus_{i\in[1;m]} \mathsf{CFG}(C_i)$. We consider several Android apps at the same time. The control flow graph of a set $\{A_1, \ldots, A_n\}$ of apps is the disjoint union of their control flow graphs: $\mathsf{CFG}(\{A_1, \ldots, A_n\}) = \biguplus_{i\in[1;n]} \mathsf{CFG}(A_i)$. The control flow graph of a set of apps is disconnected. Each connected component of $\mathsf{CFG}(\{A_1, \ldots, A_n\})$ corresponds to the control flow graph of a component in one of the apps $A_1, \ldots, A_n$. On an Android device, the components run in parallel and communicate with each other. The next challenge is thus to identify communications between the connected components in the control flow graph through IC communications.

Definition 7.2 (Communication)**.** A *communication* is a pair (m_s, m_r) of IC communication methods from ICCom such that m_r may receive the information sent by m_s.

For example, `startActivity(intent)` and `getIntent()` form a communication, as `startActivity` sends `intent` that is received by `getIntent()`. This communication relies on Intents. Another example of communication is `putInt("key", value)` and

[1] To simplify the presentation, we consider CFGs where vertices correspond to a single statement. Our approach can easily be adapted to CFGs where vertices consist of sequences of consecutive statements.

`getInt("key")`. The method `putInt()` associate `value` to `key` in a map. The value is later retrieved (by another component) through a call to method `getInt()`. A complete set of communications can be obtained from Android specifications [142].

A path in a control flow graph is a finite sequence $v_1, \ldots, v_k$ of vertices such that there is an edge $v_i \to v_{i+1}$ for every $i \in [1; k-1]$. For a path $p = v_1, \ldots, v_k$ in a CFG, let $\mathsf{firstvertex}(p)$ denotes the first vertex of the path (i.e. v_1), and let $\mathsf{lastvertex}(p)$ denotes its last vertex (i.e. v_k).

Definition 7.3 (Communication path)**.** Let $\{A_1, \ldots, A_n\}$ be a set of Android apps, and let $\mathsf{CFG}(\{A_1, \ldots, A_n\})$ be the corresponding control flow graph. A *communication path* is a sequence of path segments $p_1, \ldots, p_k$ with $k \geq 2$ such that:

- every path segment p_i is a path in $\mathsf{CFG}(\{A_1, \ldots, A_n\})$, and
- for every $i \in [1; k-1]$, $\mathsf{lastvertex}(p_i)$ and $\mathsf{firstvertex}(p_{i+1})$ form a communication.

We denote $\mathsf{App}(p_i)$ the app A_j such that p_i is a path in $\mathsf{CFG}(A_j)$, in other words, a sequence of instructions from A_j.

A communication path thus models interactions between the components of a set of Android apps. Observe that several path segments in a communication path may belong to the same component, hence to the same app.

We are now interested in communication paths that lead to a leak. We need to track sensitive information in order to detect leaks. To that purpose, we introduce a function λ that maps every expression e in Android programs to the set of sensitive source calls (hence sensitive information) required to evaluate e. We sketch a description of λ below.

For every sensitive source instruction $s \in \mathsf{SSources}$, $\lambda(s) = \{s\}$. A non-sensitive value v (literal, value obtain through a non-sensitive API call, etc) has $\lambda(v) = \emptyset$. For operators, λ is defined as the union of the calls needed to evaluate the operands, e.g. $\lambda(e_1 + e_2) = \lambda(e_1) \cup \lambda(e_2)$. Similarly for function calls, $\lambda(f(e_1, \ldots, e_n))$ is defined from $\lambda(e_1), \ldots, \lambda(e_n)$ taking into account the instructions of function f. For an assignment $x = e$, $\lambda(x) = \lambda(e)$ and $\lambda(x = e) = \lambda(e)$. The function λ can be precisely defined from the grammar of the Java language and Android APIs. Observe that a sensitive sink instruction $f(e_1, \ldots, e_n)$ of an Android app A leaks a sensitive information when $\lambda(f(e_1, \ldots, e_n))$ involves a

sensitive source call s with a permission $\gamma(s)$ that is not granted to A, i.e. $\gamma(s) \notin \mathsf{Perm}(A)$.

Definition 7.4 (Sensitive communication path)**.** A *sensitive communication path* for a set $\{A_1, \ldots, A_n\}$ of Android apps is a communication path $p_1, \ldots, p_k$ such that:

- the first instruction is a sensitive source: $\mathsf{firstvertex}(p_1) \in \mathsf{SSources}$, and,
- the last instruction is a sensitive sink: $\mathsf{lastvertex}(p_k) \in \mathsf{SSinks}$, and,
- the first instruction is needed to compute the leaked value: $\mathsf{firstvertex}(p_1) \in \lambda(\mathsf{lastvertex}(p_k))$, and,
- the last instruction is a leak: there is a sensitive source call $s \in \lambda(\mathsf{lastvertex}(p_k))$ such that $\gamma(s) \notin \mathsf{Perm}(\mathsf{App}(p_k))$.

We aim at detecting sensitive communication paths from a model of $\{A_1, \ldots, A_n\}$. Notice that not all sensitive communication paths correspond to actual collusion. Indeed, the paths in $\mathsf{CFG}(\{A_1, \ldots, A_k\})$ do not take into account the values of the variables. Thus, a sensitive communication path may not be feasible when variables are taken into account. But, every collusion corresponds to a sensitive communication path. In section 7.2, we describe *SneakLeak+*, our approach to detect app collusion. It is based on a formal verification approach to detect sensitive communication paths. We first briefly introduce formal verification.

7.1.2 Formal Verification

The process of checking whether a system (software or hardware) conforms to or violates a pre-specified set of desired properties (often referred to as the requirements) is called verification [143]. In our case, properties refer to colluding conditions. We can apply formal verification to prove the absence of colluding conditions in Android apps.

One approach for formal verification is model checking, which consists of a systematically exhaustive exploration of the mathematical model. Model-checking [144] is a powerful technique for the automated analysis of systems. It has been successfully applied to both hardware and software. An overview of the model-checking approach is depicted in Figure 7.1. It consists of modelling the program under consideration, and in specifying

the requirements in a formal language. The model represents the runs of the program, whereas the formal specification represents the set of admissible runs. A model-checking algorithm then exhaustively checks that all the runs in the model are admissible with respect to the requirements. If it is not the case, a counterexample is provided that exhibits a run and violates the requirements.

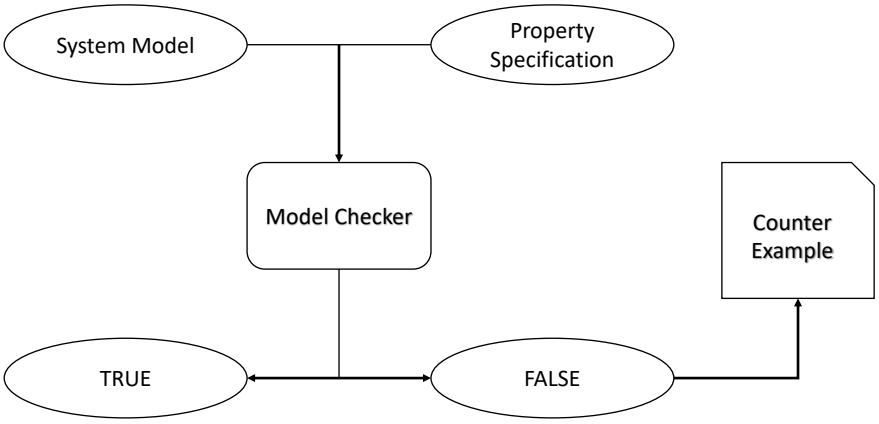


FIGURE 7.1: Model Checking Process

7.2 Proposed Approach: *SneakLeak+*

Our proposed tool *SneakLeak+* is designed to detect potential collusion (Definition 7.1). It detects sensitive communication paths (Definition 7.4). To that purpose, it identifies communication paths (Definition 7.3), as well as the flow of sensitive data along those paths. *SneakLeak+* relies on the construction of an abstract model of the apps. Static analysis is used to capture the flow of sensitive data. Finally, model-checking is used to detect sensitive communication paths that lead to a leak. In the sequel, we describe the three major phases of our approach: *Extract App Information*, *Sensitive DataFlow Analysis and Model Construction*, and *Interaction Analysis*.

7.2.1 Extract App Information

As illustrated in Algorithm 7.1, this step extracts all the essential information needed in the subsequent steps by performing static analysis. There are two main sources from where the information is extracted from app A, that are *Manifest* and bytecode (cf. Section 2.1). Line 1 will reverse engineering the app to extract its *Manifest* M and bytecode B using DARE [79]. In lines 2-3, permissions $\mathsf{Perm}(A)$ and components $\mathsf{Comp}(A)$ are extracted from the *Manifest* file. Then, in lines 4-8, program analysis and string analysis is performed using IC3 [145] which is the most precise single-app IC communications resolution tool in the literature. With this analysis, essential information are extracted for each component. We extract sensitive API calls SCalls. Sensitive API calls are identified based on the list provided by SuSi [126]. We also extract inter-process communications performed via Intents, shared preferences and content providers. ICComIn is the set of incoming IC communications which are identified by: `getExtra` attribute of Intents, `select` queries of Content Providers, or calls to methods `getInt`, `getString`, `getLong`, `getFloat` and `getBoolean` of Shared Preferences. Symmetrically, ICComOut is the set of outgoing IC communications which consist in: `putExtra` attribute of Intents, queries `insert` and `update` of Content Provider, and calls to methods `putInt`, `putString`, `putLong`, `putFloat` of Shared Preferences.

Algorithm 7.1 extractAppInfo

Input: A: Android App

Output: $\langle B, \mathsf{Perm}, \mathsf{Comp}, \mathsf{SCalls}, \mathsf{ICComIn}, \mathsf{ICComOut}\rangle$ where B is the bytecode of app A, Perm maps A to its set of permissions from $\mathsf{DangerPerms}$, Comp is the set of components in A, $\mathsf{SCalls} : \mathsf{Comp} \to 2^{\mathsf{SAPICalls}}$ maps every component to its set of sensitive API calls, $\mathsf{ICComIn} : \mathsf{Comp} \to 2^{\mathsf{ICCom}}$ and $\mathsf{ICComOut} : \mathsf{Comp} \to 2^{\mathsf{ICCom}}$ associate to every component the sets of incoming and outgoing IC communications respectively.

 # Retrieve Manifest(M) and Bytecode(B):

1: $M, B \leftarrow reverseEngineer(A)$

 # Extract permissions and components:

2: $\mathsf{Perm}(A) \leftarrow extractManifestPermissions(M)$

3: $\mathsf{Comp}(A) \leftarrow extractManifestComponents(M)$

 # Extract sensitive API calls and communications for every component in A:

4: **for all** $C \in \mathsf{Comp}(A)$ **do**

5: $\quad \mathsf{SCalls}(C) \leftarrow extractSensitiveAPICalls(C, \mathsf{DangerPerms}, B)$

6: $\quad \mathsf{ICComIn}(C) \leftarrow extractIncomingCommunications(C, B)$

7: $\quad \mathsf{ICComOut}(C) \leftarrow extractOutgoingCommunications(C, B)$

return $\langle B, \mathsf{Perm}, \mathsf{Comp}, \mathsf{SCalls}, \mathsf{ICComIn}, \mathsf{ICComOut}\rangle$

7.2.2 Sensitive DataFlow Analysis and Model Construction

A straightforward approach to detect sensitive communication paths for a set of apps $\{A_1, \ldots, A_n\}$ is to explore the control flow graph $\mathsf{CFG}(\{A_1, \ldots, A_n\})$ for such paths. Although, the control-flow graphs for average Android apps turn out to be huge. This approach would not scale up to more than a few apps. Instead, we take a different approach that consists in building a small abstraction of $\mathsf{CFG}(\{A_1, \ldots, A_n\})$. It relies on sensitive dataflow analysis and static taint analysis to extract paths in the control-flow graph, and the flow of sensitive information along those paths (the λ function in Section 7.1.1).

Algorithm 7.2 describes the construction for a single component C from $\mathsf{Comp}(A)$. The graph of app A, $\mathsf{G}(A)$, is obtained as the disjoint union of the components' graphs $\biguplus_{C \in \mathsf{Comp}(A)} \mathsf{G}(C)$. The graph $\mathsf{G}(C)$ abstracts $\mathsf{CFG}(C)$ in the sense that instructions from

Algorithm 7.2 buildComponentModel

Input: $\langle C, B, \mathsf{SCalls}, \mathsf{ICComIn}, \mathsf{ICComOut}\rangle$ where C is a component, B is the app bytecode and, SCalls, ICComIn and ICComOut associate to component C its set of sensitive API calls, its set of incoming IC communications, and its set of outgoing IC communications respectively.

Output: a finite graph $\mathsf{G}(C) = \langle V, E\rangle$ that abstracts $\mathsf{CFG}(C)$

 # Build graph vertices:

1: $V_{sources} \leftarrow \mathsf{SCalls}(C) \cap \mathsf{SSources}$
2: $V_{sinks} \leftarrow \mathsf{SCalls}(C) \cap \mathsf{SSinks}$
3: $V_{in} \leftarrow \mathsf{ICComIn}(C)$
4: $V_{out} \leftarrow \mathsf{ICComOut}(C)$
5: $V \leftarrow V_{sources} \uplus V_{sinks} \uplus V_{in} \uplus V_{out}$

 # Extract Sensitive Paths Segments:

6: **for all** vertices $(v, v') \in (V_{sources} \times V_{out}) \cup (V_{in} \times V_{out}) \cup (V_{in} \times V_{sink})$ **do**
7: **if** $sensitivePath(v, v', B)$ **then**
8: add an edge $v \rightarrow v'$ to E

return $\mathsf{G}(C) = \langle V, E\rangle$

B that are irrelevant for sensitive paths analysis are abstracted away. Furthermore, $\mathsf{G}(C)$ only preserves sensitive path segments (in the sense of Definition 7.3), but it does not preserve the structure of $\mathsf{CFG}(C)$. $\mathsf{G}(C)$ is called the path segment abstraction of C.

The vertices of $\mathsf{G}(C)$ (lines 1-5) correspond to either sensitive API calls from $\mathsf{SCalls}(C)$ (the sets $V_{sources}$ and V_{sinks}), or incoming/outgoing IC communications from $\mathsf{ICComIn}(C)$ and $\mathsf{ICComOut}(C)$ respectively (the sets V_{in} and V_{out}). The edges in $\mathsf{CFG}(C)$ correspond to path segments in the sense of Definition 7.3. These are the paths from a sensitive source to an outgoing IC communication ($V_{sources} \times V_{out}$), or from an incoming to an outgoing IC communication ($V_{in} \times V_{out}$), or from an incoming IC communication to a sensitive sink ($V_{in} \times V_{sink}$). These paths are identified using taint analysis: $sensitivePath(v, v', B)$ is true when there is a path in bytecode B from instruction v to instruction v', and such that the evaluation of v' depends on v, i.e. $v \in \lambda(v')$ (cf. Section 7.1.1). In line 6-10, an edge is added to $\mathsf{G}(C)$ for every pair of vertices linked by a sensitive path. Observe that the graph represents path fragments as edges, and that there is no path of length more than one.

7.2.2.1 Generate Collusion Model

Algorithm 7.3 shows the last step of our approach. In the previous sections, we have built an abstract model for a component in an Android app. In lines 1-8, we use

Algorithm 7.3 generateCollusionModel

Input: $\{A_1, \ldots, A_n\}$ a set of Android apps

Output: $\langle G, \rho \rangle$ where $G = (V, E)$ is a finite graph such that every communication path in $\mathsf{CFG}(\{A_1, \ldots, A_n\})$ has a corresponding path in G, and $\rho : V \to 2^{\mathsf{DangerPerms}}$ label each vertex $v \in V$ with the permissions needed to evaluate the instruction v.

 # Build the apps model:

1: **for all** $A \in \{A_1, \ldots, A_n\}$ **do**
2: $\quad \langle B, \mathsf{Perm}, \mathsf{Comp}, \mathsf{SCalls}, \mathsf{ICComIn}, \mathsf{ICComOut} \rangle \leftarrow extractAppInfo(A)$
3: $\quad$ **for all** $C \in \mathsf{Comp}(A)$ **do**
4: $\quad\quad \mathsf{G}(C) \leftarrow buildComponentModel(C, B, \mathsf{SCalls}, \mathsf{ICComIn}, \mathsf{ICComOut})$
5: $\quad \mathsf{G}(A) \leftarrow \biguplus_{C \in \mathsf{Comp}(A)} \mathsf{G}(C)$
6: $G \leftarrow \biguplus_{i \in [1;n]} \mathsf{G}(A_i)$

 # Build the communication model:

7: # let V^i_{in} and V^i_{out} respectively denote the ICComIn and ICComOut vertices in $\mathsf{G}(A_i)$
8: **for all** vertices $v_s \in V^i_{out}$ and $v_r \in V^j_{in}$ with $i \neq j$ **do**
9: $\quad$ **if** (v_s, v_r) is a communication **then**
10: $\quad\quad$ Add an edge $v_s \to v_r$ to G

 # Build the permission flow:

11: # let V^i_{source} denote the source vertices in $\mathsf{G}(A_i)$
12: $V_{source} \leftarrow \biguplus_{i \in [1;n]} V^i_{source}$
13: **for all** vertex v in G **do**
14: $\quad \rho(v) \leftarrow \begin{cases} \gamma(v) & \text{if } v \in V_{source} \\ \bigcup_{v' \to v} \rho(v') & \text{otherwise} \end{cases}$

return $\langle G, \rho \rangle$

Algorithms 7.1 and 7.2 to build a finite graph $G = (V, E)$ that is an abstract model for a set of apps $\{A_1, \ldots, A_n\}$. We now need to detect sensitive communication paths in G (cf. Definition 7.4). We reduce this problem to a reachability problem on an extension of G. As a first step, in lines 9-14, we add edges to G that correspond to communications (cf. Definition 7.2). Now, every communication path in $\mathsf{CFG}(\{A_1, \ldots, A_n\})$ is represented as a path in G. As a second step, in lines 15-19, we build a map $\rho : V \to 2^{\mathsf{DangerPerms}}$ that associate to every vertex v of G the set of dangerous permissions needed to execute the instruction v. Notice that this includes the permissions needed to evaluate the parameters of v, if any. Thus, a sensitive communication path corresponds to a path in G that starts in a source vertex v, and that ends in a sink vertex v' such that the instruction in v' requires permissions $\rho(v')$ which are not granted to its app. Notice that this implies that v and v' do not belong to the same app, hence the path takes at least one communication edge. Let $\mathsf{Perm}(v)$ denote the set of permissions granted to app A_i such that v is a vertex of $\mathsf{G}(A_i)$. Our model allows to detect sensitive communication paths in the following sense:

Lemma 7.5. *Let* $\{A_1, \ldots, A_n\}$ *be a set of Android apps, and let* $\langle G, \rho \rangle = generateCollusionModel(\{A_1, \ldots, A_n\})$. *Assuming that sensitivePath is complete*[2]*, then every sensitive communication path in* $\mathsf{CFG}(\{A_1, \ldots, A_n\})$ *has a corresponding path* $v \to \cdots \to v'$ *in* G *such that* n *is a source vertex,* n' *is a sink vertex, and* $\rho(v') \not\subseteq \mathsf{Perm}(v')$.

Owing to the path segment abstraction used to model the components, the size of apps model $\mathsf{G}(\{A_1, \ldots, A_n\})$ grows polynomially with the number of components. This allows our method to scale to big sets of apps.

7.2.2.2 Motivating Example

In this section, we provide an example of threat scenario that is possible due to Android's ICC vulnerability. In the subsequent section, we will generate a model of this example using our approach. Our threat scenario consists of two apps named *Sender* and *Receiver* as shown in Figure 7.2.

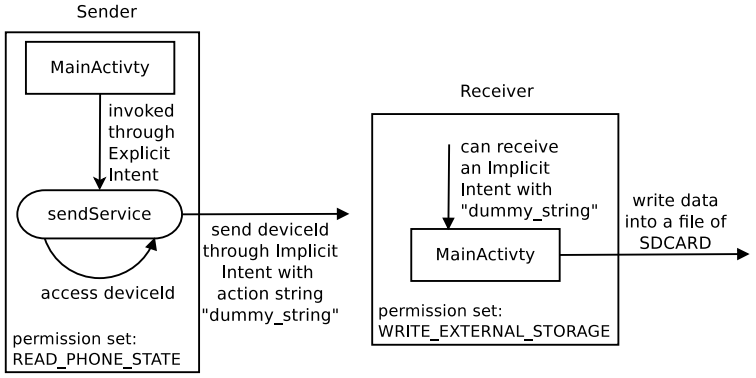


FIGURE 7.2: Potential Threat Scenario: Sender app communicates data to Receiver app through implicit intent
(Android does not check for permission privileges while passing the data)

Sender app has two components named *MainActivity* and *sendService*. Assume, `MainActivity` is invoking `sendService` component of Sender app itself through an explicit Intent. Once invoked, it calls `getDeviceId()` which is a sensitive API call that requires permission `READ_PHONE_STATE`, and this permission is mentioned in the *Manifest* file of Sender app. The output of this API call is the unique IMEI number of the device,

[2] $sensitivePath(v, v', B)$ is complete if it returns *true* whenever there is a path from instruction v to instruction v' in bytecode B.

which is then encapsulated in an implicit intent with action string *"dummy_string"* and sent. Any activity of any app installed on the device that has mentioned *"dummy_string"* in the `<intent-filter>` tag inside *Manifest* file can receive this intent.

Assume, MainActivity of Receiver app can take the intent with action string *"dummy_string"*. Therefore it can get the access to the sensitive information. Here, Receiver app does not have permission to access unique IMEI number of the device, but still, it can get this information through another co-existing app via intent. This is called privacy leakage, it can lead to the threat when Receiver app sends this data outside. In our example, when the app is writing this information to an external memory card.

7.2.2.3 Collusion Analysis using Model-Checking

In this section, we describe how we implement Algorithm 7.3 using model-checking. We illustrate our approach on the threat model example from Section 7.2.2.2. Listing 7.1 shows a simplified PROMELA model built from this example. The model maintains a boolean flag `collusion` (line 12) that becomes true when collusion is detected, and that remains false otherwise.

We model communication edges and permission flow from Algorithm 7.3 using communication features of PROMELA. Communication edges are modelled by inter-process communication channels. The process exchanges messages over the channels. In our model, the messages consist of the dangerous permissions needed to access sensitive information that is exchanged by the processes (`mtype` in line 2). Apps permissions Perm are defined in lines 5 and 6. The implicit intent is modelled in line 9 by a channel that carries `mtype` messages. The explicit intent in Figure 7.2 is not modelled as it is not used to exchange sensitive information.

There is one process (`proctype`) for each component. In lines 15-16, the process for component `Sender.mainActivity` is empty since the component does not exchange sensitive information. Component `Sender.sendService` is modelled in lines 19-25. It has one path segment that corresponds to reading the phone state (line 22), then sending the state using intents (line 23). The model for component `Receiver.mainActivity` also has only one path segment that consists in receiving sensitive information that requires permission `perm` (line 31), then writing this information to the SD card (lines 32). Since

writing to the SD card is a sensitive sink API call, it may lead to a leak. So, we check in lines 33-41 that permission `perm` is among the permissions of app `Receiver`. If not, a sensitive communication has been detected, and collusion is reported (line 40).

```
1  /* messages = DangerPerms, NAP=Not-A-Permission (service message) */
2  mtype { READ_PHONE_STATE, WRITE_EXTERNAL_STORAGE, NAP};
3
4  /* permissions of applications Sender and Receiver, i.e. Perm(Sender) and Perm(Receiver) */
5  mtype Sender_PermSet[2] = {READ_PHONE_STATE, NAP};
6  mtype Receiver_PermSet[2] = {WRITE_EXTERNAL_STORAGE, NAP};
7
8  /* intent modelled as a communication channel (handshake) */
9  chan com_example_collector_implicit = [0] of {mtype};
10
11 /* collusion detection flag (0 means false, other means true) */
12 bool collusion = 0;
13
14 /* process modelling component Sender mainActivity, i.e. G(Sender.mainActivity) */
15 proctype Sender.mainActivity() {
16 }
17
18 /* process modelling component Sender.sendService, i.e. G(Sender.sendService) */
19 proctype Sender.sendService() {
20 mtype perm;
21 do
22 :: source_read_phone_state: // sensitive source
23 com_example_collector_implicit ! READ_PHONE_STATE;
24 od
25 }
26
27 /* process modelling component Receiver.mainActivity, i.e. G(Receiver.mainActivity) */
28 proctype Receiver.mainActivity() {
29 mtype perm;
30 do
31 :: com_example_collector_implicit ? perm; -> {
32 sink_write_SD_card:          // sensitive sink
33 // check that perm belongs to Perm(Receiver)
34 for (i : 0..1) {
35 if
36 :: perm == Receiver_PermSet[i] -> goto ok;
37 :: else -> skip
38 fi
39 };
40 collusion = 1;   // collusion detected
41 ok:    skip;
42 }
43 od
44 }
```

LISTING 7.1: Interaction model for example in Figure 7.2 (simplified model)

A PROMELA model describes a set of runs. The model in Listing 7.1 only has one run. It consists in executing line 23 in process `Sender.sendService` synchronously with line 31 in process `Receiver.mainActivity`, then executing lines 34 to 40, and detecting a leak since permission `READ_PHONE_STATE` is not granted to app `Receiver`. Sensitive communication paths, hence app collusion, are detected by checking if the boolean flag

`collusion` may be set to 1. This is achieved by a standard reachability check on the model using the model-checker SPIN.

7.2.3 Incremental analysis

SneakLeak+ uses an incremental approach to scale up collusion analysis to sets of apps that can be found on an average smartphone. Algorithm 7.4 shows how to add apps incrementally during analysis. It relies on a straightforward modification of Algorithm 7.2.2.3 (line 3) that builds the communication model and the permission flow incrementally.

Algorithm 7.4 incrementalAnalysis

Input: $\{A_1, \ldots, A_n\}$ a set of Android apps
Output: "collusion" if a sensitive communication path is found, "no collusion" otherwise
1: $\langle G, \rho \rangle \leftarrow generateCollusionModel(\{A_1\})$
2: **for all** $i \in [2; n]$ **do**
3: extend $\langle G, \rho \rangle$ with $\mathsf{G}(A_i)$
4: **if** G has a reachable vertex v s.t. $\rho(v) \not\subseteq \mathsf{Perm}(v)$ **then return** "collusion"
return "no collusion"

7.3 Evaluation

To access the effectiveness of our proposed tool *SneakLeak+*, we conducted experiments to address following research questions:

RQ1 What is the motivation for interaction analysis?

RQ2 How does *SneakLeak+* perform compare to state-of-the-art inter-app vulnerability detection tools (such as COVERT, IccTA+APKCombiner, DIALDroid) on the set of benchmark apps?

RQ3 What is the overall accuracy of *SneakLeak+* in detecting inter-app vulnerabilities?

RQ4 How *SneakLeak+* can scale to perform interaction analysis of thousands of real-world Android apps?

7.3.1 Need of Interaction Analysis

To motivate the need for interaction analysis, we conducted an experiment by downloading the Device ID[3] app from Google Play. The app has permission to retrieve Device info/ID, local IP, and MAC addresses. We analyzed the app on single-app analysis tools, and results are shown in column(2) of Table 7.1. The app requests for six dangerous permissions and consequently the risk score value is moderately high.

As a second step of our experiment, we developed the similar app by reducing the permission from 6 to 1, more specifically, we retained only `READ_PHONE_STATE` permission, which is needed to get all the required details and added one more permission required to write on the external storage, i.e., `WRITE_EXTERNAL_STORAGE`. We have also added a leakage code that writes the device id info to a file named `leak.txt` and stores it on the SD card. We then performed the same analysis for the revised app. The analysis results of this app on the same tools are shown in column(3) of Table 7.1. Surprisingly, the score reduced by 60%. This demonstrates that the number of dangerous permissions required by the apps is the key feature to calculate the risk possessed by the app.

Single-App Analysis Approaches	**Device ID** (2)	**DeviceId_Collector** (3)	**DeviceId_Service** (4)	**Collector** (5)
VirusTotal (Web-Service)	1/56	0/56	0/56	0/56
VisualThreat (Web-Service)	30*	12*	7*	7*
SandDroid (Web-Service)	28*	12*	6*	6*
Permission Checker (Android App)	PersonalInfo Leakage	PersonalInfo Leakage	SAFE	SAFE
Permission Friendly (Android App)	700**	300**	100**	120**
IccTA (Open-Source Tool)	No Source No Sink	No Source No Sink	No Source No Sink	No Source No Sink
DroidSafe (Open-Source Tool)	†	Source and sink	No Source No Sink	No Source No Sink
FlowDroid (Open-Source Tool)	No Source No Sink	No Source No Sink	No Source No Sink	No Source No Sink
HelDroid (Web-Service)	†	Malicious	Benign	Benign

TABLE 7.1: Results of single-app analysis approaches on *Device ID* app and its variants $*$ = Risk Score (Scale 0-100) lesser is better, $**$ = Risk Score (Scale 0-1000) lesser is better, † = Incompatibility issues.

[3]https://play.google.com/store/apps/details?id=com.evozi.deviceid&hl=en

Some of the approaches have detected the leakage therefore as a third step, we have divided the leakage path across two apps. Sensitive information is accessed via the first app, and it is leaked to the file via a second app. This corresponds to the example in Figure 7.2. Column (4) and (5) show the results of the analysis for both apps respectively. None of the approaches can detect the leakage path as they rely on single app analysis. This brings us to the conclusion that single-app analysis is not sufficient to detect leakage paths that are distributed across multiple apps. Android framework does not check if an app that is accessing the permission-protected resource through another app has itself requested that permission. Each colluding app only needs to request a minimal set of permissions, that may make it appear benign to most of the techniques. Therefore, there is a need of interaction-app analysis that can identify such paths.

7.3.2 Comparison with the State-of-the-art Approaches

We conducted our experiments on virtual machine running Ubuntu 16.04 LTS with 16 core Intel(R) Xeon(R) E5-2699 v3 2.30GHz CPU, 360GB RAM. We evaluated both real-world apps from Google Play and benchmark apps from DroidBench and ICC-Bench. We analyzed two branches of DroidBench that contain apps with inter-app communications, namely *develop* (DroidBench 3.0) [16] and *iccta* (DroidBench 2.0) [146]. In addition, we have also released a set of 64 apps exhibiting inter-app communication for comparing the detection capabilities for collusive data leaks[4].

COVERT [12], IccTA [26]+APKCombiner [147] and DIALDroid [70] proposed inter-app vulnerability detection techniques. Table 7.2 shows comparison of these techniques with *SneakLeak+* on DroidBench, self-made apps and ICC-Bench [148] datasets. As compared to four tools, *SneakLeak+* outperforms with the highest precision (100%), highest recall (93.3%) and highest F-measure (0.97). COVERT reported highest false positives due to the inaccurate mapping of sensitive data with intent and implicit intent resolution. If any string operation is performed on sensitive data or intent, COVERT loses that occurrence and misses leakage paths. It cannot detect any leakage paths in DroidBench and our new benchmark dataset apps, hence having relatively low recall value(33.3%). IccTA+APKCombiner performs poorly among all the tools. It crashes

[4]We have contributed the set of 64 apps to DroidBench

Source App	Destination App	COVERT	IccTa+APK Combiner	DIALDroid	*SneakLeak+* (Ours)
DroidBench 3.0					
SendSMS	Echoer	○	○	●	●
StartActivityForResult1	Echoer	○	○	●	●
DeviceId_Broadcast1	Collector	○	†	●	●
DeviceId_ContentProvider1	Collector	○	†	●	●
DeviceId_OrderedIntent1	Collector	○	†	●	●
DeviceId_Service1	Collector	○	†	○	●
Location1	Collector	○	†	●	●
Location_Broadcast1	Collector	○	†	●	●
Location_Service1	Collector	○	†	○	●
Incorrect app pairings		(172) Ⓧ	‡		
Call Logs to SD Card through Implicit Intent (Self Made)					
Task21	Service21	○	†	○	●
Twin2	CallWritingImplicit	○	†	○	●
Twin7	SdReceiverimplicit	○	†	○	●
Twin10	ReceiverEx	○	†	○	●
Task14	Service14	○	†	○	●
Task15	ServiceImp	○	†	○	●
Task17	ReceiverSd	○	†	○	●
Task20	ActivitySdImp	○	†	○	●
ReflectedTask21	Service21	○	†	○	○
ReflectedTwin2	CallWritingImplicit	○	†	○	○
Incorrect app pairings			‡	(2) Ⓧ	
DroidBench (IccTA branch)					
startActivity1_source	startActivity1_sink	●	●	●	●
startService1_source	startService1_sink	●	●	●	●
startbroadcast1_source	startbroadcast1_sink	●	●	●	●
Incorrect app pairings		(104) Ⓧ	‡		
ICC-Bench					
implicit_action	implicit_src_sink	●	†	●	●
implicit_action	implicit_nosrc_sink	●	○	●	●
implicit_mix1	implicit_mix2	○	†	●	●
implicit_src_nosink	implicit_src_sink	●	○	●	●
implicit_src_nosink	implicit_nosrc_sink	●	○	●	●
implicit_src_nosink	implicit_action	●	○	●	●
implicit_src_sink	implicit_action	●	†	●	●
implicit_src_sink	implicit_nosrc_sink	●	○	●	●
Incorrect app pairings		(47) Ⓧ	‡		
Sum, Precision, Recall and F measure					
True Positive (●), higher is better		10	3	18	28
False Positive (⊗), lower is better		323	0‡	2	0
False Negative (○), lower is better		20	9	12	2
Precision, p = ●/(●+⊗)		**3.0%**	**100%‡**	**93.3%**	**100%**
Recall, r = ●/(●+○)		**33.3%**	**25%**	**60%**	**93.3%**
F-measure = 2pr/(p+r)		**.06**	**.40**	**.75**	**.97**

● correct detection Ⓧ false warning ○ missed detection † tool crashed ‡ cannot say

TABLE 7.2: Comparison of existing inter-app data leakage detection techniques

or misses the detection on most of the apps. It reported detection only on DroidBench-IccTA dataset because it is developed by the same authors. It has the lowest recall value (25%). DIALDroid performs reasonably well with precision (93.3%) and recall (60%). The reason for false positive detection in DIALDroid is the use of regular expressions for string search that checks only the occurrence of sub-string. The reason for false negative is incomplete string handling. For example, in self-made apps, sensitive data is serialized and hence detection is evaded by DIALDroid. It also fails to recognize leaks via intermediate components. All the approaches are based on static analysis including *SneakLeak+* are restricted by anti-static-analysis techniques such as reflection, dynamic code loading, and obfuscation. They make the call to $sensitivePath(v, v', B)$ in Algorithm 7.2 incomplete. Thus all static analysis based approaches miss the detection of reflected apps.

For completeness, we have also compared inter-app analysis runtime of existing state-of-the-art on 98 randomly selected apps from Google Play. COVERT does not scale for 4,753 pairs and hence crashed after 37 hours, IccTA+APKCombiner can combine only 851 pairs and took 403 hours. DIALDroid took 21 hours to complete the analysis whereas *SneakLeak+* took 18.5 hours to complete.

7.3.3 Performance and Timing

SneakLeak+ performance analysis is done by analyzing preprocessing time and interaction analysis time on different benchmark datasets and 30 popular apps from Google Play. As illustrated in Table 7.3, a large proportion of the total analysis time is required by preprocessing phase. But, each app has to undergo this phase only once.

There is a significant variation of time in benchmark apps, and Google play apps. Benchmark apps are developed to demonstrate the attack scenario and therefore contain limited data flows and $\sim$2 components per app on an average. Therefore, there are very few states in the interaction model of benchmark apps set. Whereas, real-world apps are larger in size (LOC) and composed of complex functionalities that increase preprocessing time. Similarly, real-world apps composed of $30 - 400$ components resulting in a larger interaction model.

App Name	Preprocessing Time(in Min.)	Interaction Analysis Time (in Min.)
DroidBench3.0 (11 apps)	4.4	0.13
ICC-Bench (24 apps)	10.04	0.33
Self-Made (56 apps)	25.1	1.01
DroidBench-IccTA (95 apps)	39.6	3.07
Google Play (30 apps)	135	6.75

TABLE 7.3: Performance and Timing Analysis of *SneakLeak+*

7.3.4 Scalability

In the interest of scalability, *SneakLeak+* limits the number of states to represent an app. The states represent only those components that are involved in sensitive information related communication (in/out). It has been observed that only a small proportion of communication channels are sensitive. If the component is isolated or dealing with non-sensitive data, we are not considering it into our model.

It is very important to construct small models as we did to allow the technique to scale up to a reasonable set of apps. A smartphone has an average of 30 installed apps and as shown in Table 7.3, it will take around 142 minutes to analyze the entire device. We have also performed large-scale detection of app collusion on over $11,000$ apps and identified many real-world collusion apps that are leaking private information. Therefore, it can be clearly stated that our state model can scales well on an average set of apps as compared to other state-of-the-art techniques.

7.4 Summary

In this chapter, we have presented *SneakLeak+* that enacts interaction-app analysis to detect inter-app ICC based privacy leak vulnerability. Our approach employs static analysis to represent an app into a compact form suited for formal verification. The formal analysis engine along with the threat specification is used to verify whether the potential inter-app ICC communication is safe to the user (no privacy leak). The same technique can be easily adapted to check privilege escalation instead of leaks. Like most of the static analyzers, our approach is also limited by anti-static-analysis techniques

such as reflection, dynamic code loading, and obfuscation. Experimental results, on a large-scale dataset demonstrate that *SneakLeak+* achieves the highest precision (100%), highest recall (93.3%) and highest F-measure (0.97) as compared with existing state-of-the-art approaches. Besides better accuracy, our analysis detected real-world colluding apps that are leaking private information.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this Thesis, we present analysis techniques for intra-app and inter-app for Android platform. In Part-I of the Thesis, we propose three novel analysis techniques to detect "single malicious app" which complement each other to improve analysis coverage. In the first step, we propose 'DRACO' that analyzes features extracted from the *Manifest* and code file of the app. Based on these features, classifier classify the app as malicious or benign. This technique provides insight of on-device analysis mechanism.

On-device analysis can produce the first level of warnings only due to computational restrictions. To increase the code coverage and detection accuracy, server module needs to be equipped with off-device techniques. Therefore, in the second step, we propose 'SWORD' that encapsulates the semantics of Android apps using Asymptotic Equipartition Property (AEP) which are further quantified to detect the malicious apps.

User's personal information is new cash commodity in our times. It has been observed that capturing the behaviour of apps with regard to privacy is an important factor to differentiate malicious apps from benign. To analyze such data flow paths, we propose, 'FlowMine' that considers the behaviour of an app towards sensitive information. It works on the principle that if the behaviour of a large number of benign and malicious apps are already known, then this information can be used as a metric to classify an unknown sample as benign or malicious. The frequency of occurrence of a source-sink pair across a number of malicious and benign apps is obtained. It will determine if

this pair can be used as a discriminant between malicious and benign behaviour. Each source-sink pair is assigned a rank, which is indicative of its discrimination capability.

The phenomena of information leakage pose a significant risk to the privacy of Android device users. Till now, the focus is towards single-app analysis, however, malicious app developers may create obscured leakage path scattered across multiple apps, making detection more complex problem with a challenge to detection. Having considered this challenge, in Part-II of the Thesis, we have extended our proposal to detect inter-app data leakage paths. In the first step, we proposed an approach that represents apps communication through 'application automaton' and identifies policies based on the violation of permission model through 'policy automaton.' The intersection of the two automata will detect the presence of collusion between two apps. But, for more than two apps, the search space posed by possible combinations of apps is exponential. Effective methods are needed to narrow down the search to collusion candidates of interest.

To allow large-scale multi-app analysis by reducing the search space posed by the combinations of apps, we propose 'SneakLeak+' that models app representing potential leaks only. It statically analyzes the reverse engineered intermediate code of each app, extract security relevant information, and represent the extracted information into a compact form suitable for formal verification. The formal analysis engine is used to verify the presence/absence of potential inter-app communication-based leakage in a reasonable time frame.

At present, there is no standard app dataset available to verify efficacy and scalability of methods dealing with collusion detection. Therefore, we developed 64 wide-ranging apps exhibiting collusion as our benchmark dataset. Now, this set is available as open-source at DroidBench [16] and is currently used widely. Details of some of these apps are provided in Appendix A.

8.2 Pointers to Future Work

In this Thesis, we attempted to cover research gap identified through the systematic literature review described in Chapter 2, including the development of techniques that allows single and multi-app analysis. As an avenue for the future research, we propose investigating the following extensions:

- Obfuscation and reflection pose an important challenge for program analysis [149]. So, investigating techniques to identify vulnerabilities due to these programming paradigm can improve the code coverage of analysis.

- The deviation of app's behaviour from its description that is provided by the developer is an important indicator to judge the security threats posed by the app. This deviation can be calculated by extracting features, semantics and data flow paths of the app.

- The technique proposed for multi-app analysis may have false positives as there are paths that exist but may never execute in real. For this, *Runtime Verification* based collusion detection techniques will be a helpful and possible direction of work.

Finally, since the theoretical contribution of intra and inter app analysis for Android platform is applicable to other mobile OS also. Thus, one needs to investigate the applicability of the approaches, presented in this Thesis, to other platforms.

We conclude this Thesis in the hope that it can contribute a step in the direction of securing mobile apps. We recommend users to analyze the apps before using as the app developers may not be aware of prevailing security threats and may leave some loose ends intentionally/unintentionally that can compromise user's security.

8.3 Publications

1. Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti "Android inter-app communication threats and detection techniques". Computers & Security 70-2017, 392-421.

2. Shweta Bhandari, Rishabh Gupta, Vijay Laxmi, MS Gaur, Akka Zemmari. "DRACO:DRoid Analyst Combo An Android Malware Analysis Framework", in Proceedings of the 8th International Conference on Security of Information and Networks, pp. 283-289. ACM 2015, Sochi/Russia

3. Shweta Bhandari, Rekha Panihar, Smita Naval, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur. "SWORD: Semantic AWare AndrOid MalwaRe Detector." Journal of Information Security and Applications 42 (2018): 46-56.

4. Lovely Sinha, Shweta Bhandari, Parvez Faruki, Manoj Singh Gaur, Vijay Laxmi, Mauro Conti. "FlowMine: Android App Analysis via Data Flow", in Proceedings of the 13th Annual IEEE Consumer Communications & Networking Conference (CCNC), 2016, pp 435-441, Las Vegas, USA.

5. Shweta Bhandari, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur. "Intersection Automata based Model for Android Application Collusion", in Proceedings of the 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2016, pp 901-908, Crans-Montana, Switerzland.

6. Shweta Bhandari, Frederic Herbreteau, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, and Partha S. Roop. "POSTER: Detecting Inter-App Information Leakage Paths." in Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS) 2017, pp 908-910, Abu Dhabi, UAE.

7. Shweta Bhandari, Frederic Herbreteau, Vijay Laxmi, Akka Zemmari, Partha S. Roop, and Manoj Singh Gaur. "SneakLeak: Detecting Multipartite Leakage Paths in Android Apps." In Trustcom/BigDataSE/ICESS, August 1-4, 2017, Sydney Australia IEEE, pp. 285-292.

8. Shweta Bhandari, Frederic Herbreteau, Vijay Laxmi, Akka Zemmari, Partha S. Roop, and Manoj Singh Gaur. "SneakLeak+: Large-Scale

Klepto Apps Analysis" Future Generations Computer Systems (2018). https://doi.org/10.1016/j.future.2018.05.047

Appendices

Appendix A

Benchmarking Colluding Apps for Analysis: DroidBench 3.0

DroidBench [16] is an open benchmark suite developed and maintained by a research group at Technical University Darmstadt, Germany. It contains toy Android apps that exhibit challenges for data flow analysis tools. Android security research community uses this benchmark suite to verify the effectiveness and accuracy of analysis tools. Table A.1 shows all the releases of DroidBench till now.

Version	Year
3.0-develop	-
2.0	Jan 23, 2015
1.1	Jul 6, 2013
1.0	May 8, 2013

Table A.1: DroidBench Releases

DroidBench 2.0 [138] is the collection of 174 testcases under 19 categories. DroidBench 3.0 [16] consists of 190 testcases. The samples are donated by different international research groups all over the globe. Table A.2 shows some of the research work that uses DroidBench apps for evaluating their proposed technique.

Title	Place	Year
MalDroid [150]	AsiaCCS	2017
DIALDroid [150]	AsiaCCS	2017
SneakLeak [151]	TrustCom	2017
Detecting Inter-App Leakage [67]	AsiaCCS	2017
TEE [150]	ICSE	2016
AspectDroid [152]	CODASPY	2016
IACDroid [153]	ICISA	2016
Harvester [154]	NDSS	2016
IntelliDroid [65]	NDSS	2016
PIFT [155]	ASPLOS	2016
COVERT [12]	IEEE TSE	2015
EdgeMiner [156]	NDSS	2015
IccTA [157]	ICSE	2015
DESCRIBEME [158]	CCS	2015
DroidSafe [159]	NDSS	2015
FlowDroid [160]	ACM SIGPLAN	2014
Epicc [135]	USENIX	2013

TABLE A.2: Research Work uses Benchmarking Apps for Evaluation

A.1 Inter-App Communication Category

Inter-App Communication is one the categories that contains apps exhibiting sensitive information leakage through multiple apps (collusion). DroidBench 2.0 has three apps involved in inter-app communication through activity component. Due to lack of availability of benchmark apps that exhibits collusion, we created our own dataset and made them available as open-source which can be used for the assessment of tools to detect privacy leakage through collusion.

We developed 64 new apps that are diverse in the components used for communication, and type of Intent based communication channels (implicit, explicit, ordered). Out of which 8 are the part of DroidBench 3.0. These samples perform inter-app communication through intents using services, activities, broadcast receivers, content providers, implicit intents, explicit intents and ordered intents. These apps access sensitive information and send it to the Collector app where the data is leaked. Tools like Epicc[135], IccTA[157], FlowDroid[160], DroidSafe[159] and AmanDroid[63] miss the detection of leakage as they

perform single app analysis. A detailed description of the samples is presented in the subsequent sections.

A.1.1 DeviceId Leakage Apps

- **DeviceId_Broadcast1**: The device id is sent to a broadcast receiver and from there on to the collector app.
- **DeviceId_ContentProvider1**: The device id is stored in a content provider and, independent from the content provider, sent to the Collector app.
- **DeviceId_OrderedIntent1**: The device id is obtained and sent to a broadcast receiver in the current app. There are multiple broadcast receivers with different priorities. Only the higher-priority receiver relays the data to the Collector app, the lower-priority receiver only shows the data to the user (no leak).
- **DeviceId_Service1**: This app starts a service which sends the device id to the Collector app where it is leaked.
- **Location1**: This app obtains the location data and sends it to the Collector app.

A.1.2 Location Information Leakage Apps

- **Location1**: This app obtains the location data through an activity which then sends the data to the Collector app.
- **Location_Broadcast1**: This app obtains the location data, and sends it to a broadcast receiver in the same app. This broadcast receiver then sends the data to the Collector app.
- **Location_Service1**: This app obtains the location data, and sends it to a service in the same app. This service then sends the data to the Collector app.

A.1.3 Sink App: Collector

- **Collector**: The data received through an intent is written into a file on the SD card.

Appendix B

Brief Bio-Data

Shweta Bhandari received the Masters of Technology degree in Computer Science from the Devi Ahilya University, Indore, in 2013. She is a research scholar at Malaviya National Institute of Technology Jaipur, India in the Department of Computer Science and Engineering under the supervision of Prof. Manoj Singh Gaur, MNIT Jaipur, CSE Department, Prof. Vijay Laxmi, MNIT Jaipur, CSE Department and Dr. Akka Zemmari, Bordeaux Laboratory of Research in Computer Science, University of Bordeaux, Talence, France. Her research work focuses on sensitive information exfiltration through Android apps. She is also a part of Security Analysis Framework in Android Platform Project (Grant:1000109932) funded by Ministry of Electronics and Information Technology (MeitY), Government of India.

Bibliography

[1] Report. http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. [Online; accessed 15-November-2017].

[2] Report. https://www.idc.com/promo/smartphone-market-share/os. [Online; accessed 15-November-2017].

[3] OWASP Mobile Security Checklist. https://github.com/OWASP/owasp-mstg/blob/master/Checklists/Mobile_App_Security_Checklist-English.xlsx. [Online; accessed 10-November-2018].

[4] News. https://gadgets.ndtv.com/apps/news/apps-android-google-play-malware-infected-sms-charges-premium-fake-services-1750785. [Online; accessed 18-January-2018].

[5] Blog. https://www.kaspersky.com/blog/cloak-and-dagger-attack/16960/. [Online; accessed 17-November-2017].

[6] FBI. Android malware slembunk and marcher actively target us financial institutions' customers [online]. https://info.publicintelligence.net/FBI-SlemBunkMalware.pdf, May 2016. [Online; accessed 19-January-2018].

[7] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[8] Norm Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[9] Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM Journal of Research and Development*, 57(6):10–1, 2013.

[10] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277. ACM, 2012.

[11] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, and Mauro Conti. Android app collusion threat and mitigation techniques. *arXiv preprint arXiv:1611.10076*, 2016.

[12] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. volume 41, pages 866–886. IEEE, 2015.

[13] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, 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. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[14] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the USENIX Security Symposium*, volume 2, 2011.

[15] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, 2014.

[16] DroidBench 3.0. `https://github.com/secure-software-engineering/DroidBench/tree/develop`. [Online; accessed 02-September-2016].

[17] Sensitive Data. `https://play.google.com/about/privacy-security-deception/personal-sensitive/`. [Online; accessed 01-December-2017].

[18] APK. `https://en.wikipedia.org/wiki/Android-application-package`. [Online; accessed 21-February-2016].

[19] Signing. `http://developer.android.com/tools/publishing/app-signing.html`. [Online; accessed 01-January-2016].

[20] Andre Egners, Ulrike Meyer, and Bjorn Marschollek. Messing with android's permission model. In *Proceedings of the IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*, pages 505–514, Washington, DC, USA, 2012. IEEE Computer Society.

[21] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*, pages 40–49. IEEE, 2014.

[22] SandBoxing. `http://developer.android.com/training/articles/security-tips.html`. [Online; accessed 01-January-2016].

[23] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 627–638, New York, USA, 2011. ACM.

[24] Permissions. `http://developer.android.com/guide/topics/security/permissions.html`. [Online; accessed 21-February-2016].

[25] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE security & privacy*, 1:50–57, 2009.

[26] Li Li, Alexandre Bartel, Tegawendé François D Assise Bissyande, 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 IEEE International Conference on Software Engineering (ICSE)*, 2015.

[27] OWASP Mobile Checklist Final 2016 . `https://drive.google.com/file/d/0BxOPagp1jPHWYmg3Y3BfLVhMcmc/view`. [Online; accessed 02-March-2016].

[28] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, volume 14, pages 23–26, 2014.

[29] Grant J Smith. *Analysis and Prevention of Code-Injection Attacks on Android OS*. PhD thesis, University of South Florida, 2014.

[30] Darell JJ Tan, Tong-Wei Chua, Vrizlynn LL Thing, et al. Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 47(4):58, 2015.

[31] The AndroidManifest.xml File. `http://lyle.smu.edu/~coyle/cse7392mobile/handouts/s01.The%20AndroidManifest.pdf`. [Online; accessed 21-May-2015].

[32] Henrik Søndberg Karlsen, Erik Ramsgaard Wognsen, Mads Chr Olesen, and René Rydhof Hansen. Study, formalisation, and analysis of dalvik bytecode. In *Informal Proceedings of the 7th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*. Citeseer, 2012.

[33] Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input dependence analysis via graph reachability. In *Proceedings of 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 25–34. Sun Microsystems, Inc., 2008.

[34] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[35] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Proceedings of the 3rd World Congress on Software Engineering (WCSE)*, pages 101–104. IEEE, 2012.

[36] Analyzing Data flow. `https://www.jetbrains.com/help/idea/2016.1/analyzing-data-flow.html`. [Online; accessed 25-April-2016].

[37] Suzanna Schmeelk, Junfeng Yang, and Alfred Aho. Android malware static analysis techniques. In *Proceedings of the 10th Annual Cyber and Information Security Research Conference (CISR)*, pages 5:1–5:8, New York, USA, 2015. ACM.

[38] 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 *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[39] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *ACM Sigplan Notices*, volume 37, pages 57–68. ACM, 2002.

[40] David Callahan. *The program summary graph and flow-sensitive interprocedual data flow analysis*, volume 23. ACM, 1988.

[41] Eugene M Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–230. ACM, 1981.

[42] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[43] Antonia J Spyridi and Aristides AG Requicha. Accessibility analysis for the automatic inspection of mechanical parts by coordinate measuring machines. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1284–1289. IEEE, 1990.

[44] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android

malware. In *Proceedings of the 7th European Workshop on System Security*, page 5. ACM, 2014.

[45] Droidbox. `http://code.google.com/p/droidbox/;`. [Online; accessed 10-October-2015].

[46] Valerio Costamagna and Cong Zheng. Artdroid: A virtual-method hooking framework on android art runtime. pages 24–32, 2016.

[47] Suhas Gupta, Pranay Pratap, Huzur Saran, and S Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *Proceedings of the International Workshop on Dynamic systems analysis*, pages 65–72. ACM, 2006.

[48] Abhinav Pathak, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the 6th Conference on Computer Systems*, pages 153–168. ACM, 2011.

[49] Pedro Machado, José Campos, and Rui Abreu. Mzoltar: automatic debugging of android applications. In *Proceedings of the International Workshop on Software Development Lifecycle for Mobile*, pages 9–16. ACM, 2013.

[50] Lok-Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceeding of the USENIX Security Symposium*, pages 569–584, 2012.

[51] Mingshen Sun, Min Zheng, John Lui, and Xuxian Jiang. Design and implementation of an android host-based intrusion prevention system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 226–235. ACM, 2014.

[52] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android security framework: Enabling generic and extensible access control on android. *arXiv preprint arXiv:1404.1395*, 2014.

[53] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. *Nicta*, 2013.

[54] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 252–261. IEEE, 2011.

[55] Peter Szor. *The art of computer virus research and defense*. Pearson Education, 2005.

[56] William Enck, Machigar Ongtang, and Patrick McDaniel. Mitigating android software misuse before it happens. Citeseer, 2008.

[57] Rubin Xu, Hassen Saïdi, and Ross J Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the USENIX Security Symposium*, volume 2012, 2012.

[58] William Enck. Defending users against smartphone apps: Techniques and future directions. In *Proceedings of the Information Systems Security*, pages 49–70. Springer, 2011.

[59] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Proceedings of the International Conference on Information Security*, pages 331–345. Springer, 2010.

[60] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the USENIX Security Symposium*, pages 131–146, 2013.

[61] OWASP. `https://www.owasp.org/index.php/Access_Control_Cheat_Sheet`. [Online; accessed 10-March-2017].

[62] A Ubale Swapnaja, G Modani Dattatray, and S Apte Sulabha. Analysis of dac mac rbac access control based models for security. *Analysis*, 104(5), 2014.

[63] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[64] Irina Mariuca Asavoae, Jorge Blasco, Thomas M Chen, Harsha Kumara Kalutarage, Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach, and Siraj Ahmed Shaikh. Towards automated android app collusion detection. In *Proceedings of the Workshop on Innovations in Mobile Privacy and Security (IMPS)*, pages 29–37, 2016.

[65] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[66] Shweta Bhandari, Vijay Laxmi, Akka Zemmari, and Manoj Singh Gaur. Intersection automata based model for android application collusion. In *Proceedings of the 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 901–908. IEEE, 2016.

[67] Shweta Bhandari, Frederic Herbreteau, Vijay Laxmi, Akka Zemmari, Partha S Roop, and Manoj Singh Gaur. Poster: Detecting inter-app information leakage paths. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 908–910. ACM, 2017.

[68] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, USA, 2014. ACM.

[69] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 118–128, New York, USA, 2015. ACM.

[70] Amiangshu Bosu, Fang Lio, Danfeng Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 71–85. ACM, 2017.

[71] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2014.

[72] Fang Liu, Haipeng Cai, Gang Wang, Danfeng Yao, Karim Elish, and Barbara Ryder. Mr-droid: A scalable and prioritized analysis of inter-app communication risks. In *Proceedings of the 7th Conference on Data and Application Security and privacy (CODASPY)*, 2017.

[73] Karim O. Elish, Danfeng Daphne Yao, and G. Ryder Barbara. On the need of precise inter-app icc classificationfor detecting android malware collusions. In *Proceedings of the Security and Privacy Workshops*, pages 116–127, 2015.

[74] D. Sbirlea, M.G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM Journal of Research and Development*, 57(6):10:1–10:12, November 2013.

[75] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128. ACM, 2015.

[76] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[77] Dex2Jar. `https://github.com/pxb1988/dex2jar`. [Online; accessed 10-May-2015].

[78] Damien Octeau, William Enck, and Patrick McDaniel. The ded decompiler. *Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Tech. Rep. NAS-TR-0140-2010*, 2010.

[79] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. Citeseer, 2012.

[80] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Proceeding of the Cetus Users and Compiler Infastructure Workshop (CETUS)*, 2011.

[81] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[82] APKTool. `http://ibotpeaches.github.io/Apktool/`. [Online; accessed 21-March-2016].

[83] A-D Schmidt, Rainer Bye, H-G Schmidt, Jan Clausen, Osman Kiraz, Kamer A Yuksel, Seyit Ahmet Camtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on android. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–5. IEEE, 2009.

[84] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. â€œandromalyâ€: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[85] Smali format. `https://code.google.com/p/smali`. [Online; accessed 28-April-2015].

[86] S. Mika, C. Schafer, P. Laskov, D. Tax, and K.-R. Muller. *Support Vector Machines.*

[87] Malware Genome Project. `http://www.malgenomeproject.org`. [Online; accessed 20-April-2015].

[88] Google Play Store. `http://play.google.com`. [Online; accessed 25-May-2015].

[89] Google Play App Crawler. `https://github.com/Akdeniz/google-play-crawler`. [Online; accessed 23-April-2015].

[90] Virusshare. `http://virusshare.com`. [Online; accessed 15-April-2015].

[91] Contagio Minidump. `http://contagiominidump.blogspot.com/`. [Online; accessed 23-April-2015].

[92] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[93] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[94] Virustotal. `http://virustotal.com/`. [Online; accessed 10-May-2015].

[95] Android Developers. The developer's guide. ui/application exerciser monkey, 2012.

[96] Guillermo Suarez-Tangil, Mauro Conti, Juan E Tapiador, and Pedro Peris-Lopez. Detecting targeted smartphone malware with behavior-triggering stochastic models. In *Proceedings of the European Symposium on Research in Computer Security*, pages 183–201. Springer, 2014.

[97] Thomas M Cover and Joy A Thomas. *Elements of information theory 2nd edition*. Wiley-interscience, 2006.

[98] AEPNotes. `http://www2.isye.gatech.edu/~yxie77/ece587/Lecture5.pdf`. [Online; accessed 10-March-2017].

[99] Cewei Cui, Zhe Dang, and Thomas R Fischer. Typical paths of a graph. *Fundamenta Informaticae*, 110(1-4):95–109, 2011.

[100] Xi Xiao, Zhenlong Wang, Qing Li, Shutao Xia, and Yong Jiang. Back-propagation neural network on markov chains from system call sequences: a new approach for detecting android malware with system call sequences. *IET Information Security*, 11(1):8–15, 2016.

[101] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[102] Hidetoshi Ishibashi, Sayaka Hihara, and Atsushi Iriki. Acquisition and development of monkey tool-use: behavioral and kinematic analyses. *Canadian journal of physiology and pharmacology*, 78(11):958–966, 2000.

[103] MarkovNotes. `http://people.math.aau.dk/~jm/courses/PhD06StocSim/Chapters2.4.5.pdf`. [Online; accessed 10-April-2017].

[104] Monkey Tool. `https://developer.android.com/studio/test/monkey.html`. [Online; accessed 17-January-2018].

[105] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2016.

[106] Stefan Edelkamp and Richard E Korf. The branching factor of regular search spaces. In *Proceeding of the AAAI/IAAI*, pages 299–304, 1998.

[107] Subhransu Maji, Alexander C Berg, and Jitendra Malik. Classification using intersection kernel support vector machines is efficient. In *Proceeding of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8. IEEE, 2008.

[108] WEKA. `http://www.cs.waikato.ac.nz/ml/weka/`. [Online; accessed 10-April-2017].

[109] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[110] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[111] Manuel Fernandez-Delgado, Eva Cernadas, Senen Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research*, 15(1):3133–3181, 2014.

[112] IccRE. `https://sites.google.com/site/icctawebpage/dataset`. [Online; accessed 26-May-2016].

[113] Michael J Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16(3):319–348, 1984.

[114] E. Mariconti, L. Onwuzurike, P. Andriotis, E.D. Cristofaro, G.J. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[115] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 221–233. ACM, 2014.

[116] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM, 2017.

[117] 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*, pages 15–26. ACM, 2011.

[118] Yiran Li and Zhengping Jin. An android malware detection method based on feature codes. 2015.

[119] Lok Kwong Yan and Heng Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceeding of the 21st USENIX Security Symposium*, pages 569–584, 2012.

[120] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[121] 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 *Proceeding of the IEEE Security and Privacy Workshops (SPW)*, pages 252–261. IEEE, 2016.

[122] Smita Naval, Vijay Laxmi, Muttukrishnan Rajarajan, Manoj Singh Gaur, and Mauro Conti. Employing program semantics for malware detection. *IEEE Transactions on Information Forensics and Security*, 10(12):2591–2604, 2015.

[123] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing*, 8(4):588–601, 2011.

[124] DangerousPermissions. `https://developer.android.com/guide/topics/permissions/overview.html#permission-groups`. [Online; accessed 15-February-2018].

[125] 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 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[126] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[127] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[128] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 95–109. IEEE, 2012.

[129] DroidAnalyst. `http://droidanalyst.org/`. [Online; accessed 10-May-2015].

[130] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. Android inter-app communication threats and detection techniques. *Computers & Security*, 70:392–421, 2017.

[131] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[132] Karim O Elish, Danfeng Daphne Yao, and Barbara G Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2015.

[133] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.

[134] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[135] 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*, 2013.

[136] 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*, pages 576–587. ACM, 2014.

[137] George Karakostas, Richard J. Lipton, and Anastasios Viglas. On the complexity of intersecting finite state automata and $\mathcal{NL}$ versus $\mathcal{NP}$. *Theor. Comput. Sci.*, 302(1-3):257–274, 2003.

[138] DroidBench 2.0. `https://github.com/secure-software-engineering/DroidBench`. [Online; accessed 02-June-2015].

[139] Vineeta Jain, Shweta Bhandari, Vijay Laxmi, Manoj Singh Gaur, and Mohamed Mosbah. Sniffdroid: Detection of inter-app privacy leaks in android. In *Proceedings of the 16th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*, pages 331–338. IEEE, 2017.

[140] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 217–228, 2012.

[141] Frances E. Allen. Control flow analysis. volume 5, pages 1–19, New York, USA, July 1970. ACM.

[142] KeyValue. `https://developer.android.com/training/basics/data-storage/index.html`. [Online; accessed 18-September-2017].

[143] Roopak Sinha, Parthasarathi Roop, and Samik Basu. *Correct-by-construction approaches for SoC design*. Springer, 2014.

[144] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[145] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[146] DroidBench IccTA. `https://github.com/secure-software-engineering/DroidBench/tree/iccta`. [Online; accessed 21-December-2016].

[147] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *ICT Systems Security and Privacy Protection*, pages 513–527. Springer, 2015.

[148] ICC-Bench. `https://github.com/fgwei/ICC-Bench`. [Online; accessed 21-December-2016].

[149] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, et al. * droid: Assessment and evaluation of android application analysis tools. *ACM Computing Surveys (CSUR)*, 49(3):55, 2016.

[150] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, pages 923–934. ACM, 2016.

[151] Shweta Bhandari, Frederic Herbreteau, Vijay Laxmi, Akka Zemmari, Partha S Roop, and Manoj Singh Gaur. Sneakleak: Detecting multipartite leakage paths in android apps. In *Proceedings of the 16th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*, pages 285–292. IEEE, 2017.

[152] Aisha Ali-Gombe, Irfan Ahmed, Golden G Richard III, and Vassil Roussev. Aspectdroid: Android app analysis system. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, pages 145–147. ACM, 2016.

[153] Nguyen Tan Cam, Pham Van Hau, and Tuan Nguyen. Android security analysis based on inter-application relationships. In *Information Science and Applications (ICISA) 2016*, pages 689–700. Springer, 2016.

[154] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[155] Man-Ki Yoon, Negin Salajegheh, Yin Chen, and Mihai Christodorescu. Pift: Predictive information-flow tracking. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 713–725. ACM, 2016.

[156] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[157] Li Li, Alexandre Bartel, Tegawendé François D Assise Bissyande, 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 IEEE International Conference on Software Engineering (ICSE)*, 2015.

[158] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards automatic generation of security-centric descriptions for android apps. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 518–529. ACM, 2015.

[159] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droidsafe. In *Proceedings of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[160] 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 *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.