Measuring and Mitigating Security and Privacy Issues on Android Applications

Candidate:
Lucky Onwuzurike

Supervisor:
Dr. Emiliano De Cristofaro

Examiners:
Professor Lorenzo Cavallaro, *Kings College London*
Dr. Steven Murdoch *UCL Computer Science*

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Information Security Research Group
Department of Computer Science
University College London

Declaration

I, Lucky Onwuzurike, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been appropriately indicated in the thesis.

Abstract

Over time, the increasing popularity of the Android operating system (OS) has resulted in its user-base surging past 1 billion unique devices. As a result, cybercriminals and other non-criminal actors are attracted to the OS due to the amount of user information they can access. Aiming to investigate security and privacy issues on the Android ecosystem, previous work has shown that it is possible for malevolent actors to steal users' sensitive personal information over the network, via malicious applications, or vulnerability exploits etc., presenting proof of concepts or evidences of exploits. Due to the ever-changing nature of the Android ecosystem and the arms race involved in detecting and mitigating malicious applications, it is important to continuously examine the ecosystem for security and privacy issues.

This thesis presents research contributions in this space, and it is divided into two parts. The first part focuses on measuring and mitigating vulnerabilities in applications due to poor implementation of security and privacy protocols. In particular, we investigate the implementation of the SSL/TLS protocol validation logic, and properties such as ephemerality, anonymity, and end-to-end encryption. We show that, despite increased awareness of vulnerabilities in SSL/TLS implementation by application developers, these vulnerabilities are still present in popular applications, allowing malicious actors steal users' information. To help developers mitigate them, we provide useful recommendations such as enabling SSL/TLS pinning and using the same certificate validation logic in their test and development environments. The second part of this thesis focuses on the detection of malicious applications that compromise users' security and privacy, the detection performance of the different program analysis approach, and the influence of different input generators during dynamic analysis on detection performance. We present a novel method for detecting malicious applications, which is less susceptible to the evolution of the Android ecosystem

(i.e., changes in the Android framework as a result of the addition/removal of API calls in new releases) and malware (i.e., changes in techniques to evade detection) compared to previous methods.

Overall, this thesis contributes to knowledge around Android apps with respect to, vulnerability discovery that leads to loss of users' security and privacy, and the design of robust Android malware detection tools. It highlights the need for continual evaluation of apps as the ecosystem changes to detect and prevent vulnerabilities and malware that results in a compromise of users' security and privacy.

Impact Statement

The problems we address in this thesis as well as the analysis, results, and tools we present herein, contribute to the advancement of research in the Android *appified* world and business processes. Specifically, the problem of vulnerabilities in applications (apps) as well as the dissemination of malware masked as legitimate apps impacts our daily lives due to the increasing number of always-on, always-connected devices that these apps run on. Finding solutions or ways to mitigate these problems is at the center of the antivirus industry and research dedicated to systems security. To address these problems, we conduct experiments, develop novel techniques which we implement in easy-to-use tools, and present the results in peer-reviewed publications. Thus, the work in this thesis contributes to knowledge in the research community and industry.

One of our findings is that previously known security vulnerabilities around the implementation of SSL certificate validation are still prevalent in popular apps. We believe this is as a result of developers misunderstanding the security risks behind their decisions. For example, developers who do not want to pay for certificates signed by a recognized certificate authority (CA), use self-signed certificates on their app's endpoint. When they encounter errors during test due to the Android operating system (OS) not accepting self-signed certificates, they often copy vulnerable code from online forums such as Stack Overflow. To help developers mitigate this problem, we provide publicly available code sample that does certificate validation correctly with both self-signed and CA-signed certificates.

Furthermore, the process of detecting malicious apps has developed into a cat-and-mouse game. When antivirus engines begin to detect malicious apps, the malware authors are forced to evolve their technique so as to evade detection. This, in turn, requires antiviruses to update their database (in cases where detection is based on signatures) or retrain their detection model (in cases where the detection applies machine learning). If malware

authors modify their evasion technique frequently, then antivirus engines may also need to be updated frequently – which might be costly. In this thesis, we present a novel technique for detecting malicious apps that does not require frequent update or retraining to still be effective. For example, we achieve relatively high detection performance when predicting samples one or two years newer/older than the samples used for training. Our technique which we implement as a tool called MaMaDroid, is publicly available for free and can be leveraged by antivirus developers or individuals in detecting malicious apps.

Finally, the different works that form this thesis have been published in international conference proceedings and online repositories; thus, other researchers working in this field of research can access and use them as references for their work.

Acknowledgments

I give all glory to God for how far He has brought me, His grace that is abound in me, and His mercies that has sustained me.

I thank all the people that has made my PhD journey pleasant, fruitful, and one that I will forever remember all the days of my life. First on the list is my primary supervisor, Dr. Emiliano De Cristofaro. Thank you for all the times you asked "why?" and "and so?". These questions made me introspect my research decisions and methods, and helped me to express my thoughts clearly. I also extend my gratitude to my secondary supervisor, Dr. Gianluca Stringhini for all the times you were available to answer questions and to help me formulate some of my thoughts.

A huge thank you also to every member of the UCL Information Security Group (ISG), the faculties and students especially members of the **best** office – Room 6.22. You made my time at UCL worthwhile and I am looking forward to working with all of you again. Special thanks also goes to Sebastian Meiser and Mary Maller for their feedback on selected Sections of my thesis. Also, to Jeremiah Onaolapo for all the nation building and "leadership" discussions we had and for your numerous feedback on diverse topics.

To my parents, brothers, sister, and church family, thank you for your love, encouragement, and support. To my friends, I say thank you for all the laughs, support and time together. In particular, to Nasiru "El Pistolero" Mohammed for the barbecues you organized, our summer trips, and for proofreading a chapter of my thesis. I also want to acknowledge my sponsors, Petroleum Technology Development Fund (PTDF) for their support.

Finally, to everyone I have coauthored a paper with, thank you for all the invaluable lessons on management, communication, and team work you taught me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we present the research questions this thesis addresses and summarize the research approaches we used to address them. We present a summary of our contributions, outline the thesis structure and finally, highlight the contributions of the author of this thesis to the publications that form this thesis.

In June 2015, there were 2.6 billion active smartphones worldwide, with each smartphone generating an average monthly data traffic of 1 GB [78]. By June 2016, these numbers had increased to 3.2 billion and 1.4 GB, respectively [79]. Moreover, IDC estimates that 87% of all smartphones worldwide run Android [120]. With these numbers, it is not surprising that attackers target the Android ecosystem, as exploiting possible vulnerabilities would lead to a compromise of a large user base.

As of January 2018, there were more than 3.5 million applications (apps) on the Google Play Store (the main market for Android apps) [24]. These apps are designed to meet diverse user needs that include banking, communication, social networking, gaming etc. and thus, handle both sensitive and non-sensitive user information. Cybercriminals target the mobile ecosystem on the basis of the information it handles, and they usually do so by exploiting vulnerabilities in apps developed by other developers or by designing their own apps to steal user information. For example, as of August 2016, there are 20 apps with over one billion downloads (on unique devices) [22] implying that an attacker that is able to exploit a vulnerability in any one of these apps, will have access to information they can sell at a very high price on the black market. Other non-criminal actors also target the ecosystem by using user information to serve them advertisements. They usually do

so by collecting information about users that allow them profile the users for effective ad targeting. These security and privacy issues i.e., information exfiltration and user profiling motivate the need to examine the Android ecosystem to ensure these issues are mitigated.

Overall, developments in the Android ecosystem have attracted the attention of the research community, with several research studies highlighting vulnerabilities that lead to security and privacy violations or usability issues. The security and privacy issues brought to light include: transmitting sensitive information in the clear [121], accepting invalid SSL certificates [82], permissions abuse [41], malware masked as legitimate apps [226], memory corruption [72], privilege escalation [39] etc. and they usually have varying impact depending on their severity, version of the OS and the corresponding user base, ease of implementation, and type of devices affected. In Chapter 3, we review research efforts on the Android ecosystem.

One area which has attracted the attention of the research community is the continuous increase in the number of Android malware.[1] As smartphones have become ubiquitous, they are used to carry out sensitive operations, e.g., financial transaction, or to control home appliances or medical devices. Thus, it has become profitable for malicious actors to write malware cloaked as legitimate apps that steal information from devices or hold their users to ransom. As a result, several research works have designed tools to classify an app as either malicious or benign using, e.g., the use of dangerous permissions [25, 47], sequence of calls [42], fingerprinting [43, 66], and frequency analysis to determine API calls used mostly by malware [14]. These tools are based on a program analysis type (i.e., static, dynamic, or hybrid analysis), which along with the techniques used, effects the detection performance of the tools. For example, Chen et al. [55] show that "known best-performing" malware classifier tools perform badly when tested on new malware datasets. This is moreso because, as both Android malware and the operating system itself evolve, it is very challenging to design robust malware detection tools that can operate for long

[1] `https://www.av-test.org/en/statistics/malware`

periods of time at an acceptable high detection rate without the need for modifications or costly retraining. Thus, more research work is needed to design better malware classifiers that are robust to these changes.

Another line of work focuses on the secure transfer of information over the network. Unlike web browsers, which provide visual feedback to users on the application layer transfer protocol being used, non-browser apps do not. The absence of visual feedback on non-browser apps means usage of such apps is based on trust that developers are sending confidential data securely, i.e., over SSL and in a way that is not susceptible to Man-in-the-Middle (MiTM) attacks. Thus, users are not aware that sending sensitive data via some of these apps is not secure and as a result, they do not take any step to protect themselves. This visual difference between browsers and non-browser apps that may result in different user behavior, motivates a number of research projects. These projects examined the implementation of SSL by non-browser apps and the SSL libraries these apps use, finding a number of vulnerabilities that allowed them conduct successful MiTM attacks [82, 101, 189]. Vulnerabilities exposed include trusting all certificates, accepting all hostnames for a certificate, accepting self-signed certificates, or using customized libraries that skip one or two of the certificate validation process. As it is important to transmit confidential or sensitive data such as: login credentials, files, bank details, IMEI number, geolocation etc., securely, it is crucial that researchers continuously investigate the implementation of SSL by developers and to also help them solve difficult challenges relating to implementing security protocols. Also, in an attempt to outdo competitors providing similar services, some developers advertise that their apps enhance users' security or privacy due to the protocols they implement in these apps. Thus, it is also important to analyze whether such claims match the reality of what a user experiences and whether the implementation of the protocols are void of vulnerabilities.

Although several studies have investigated the security and privacy concerns associated with Android, discovering vulnerabilities, and providing detection and mitigation tools etc.,

as time passes, new concerns arise, malicious actors evolve their techniques, *old* vulnerabilities are not fixed etc. These concerns thus motivate the need to continuously investigate the Android ecosystem to ensure they are measured and mitigated. Therefore, this thesis presents research efforts in this area.

1.1 Thesis Objectives

The main objective of this thesis is to measure and mitigate security and privacy issues on Android apps. To do so, this section first highlights the research problems we investigate, and then presents a summary of the methodology we employ to address them.

This thesis is divided into two parts. In the first part, we measure and mitigate security and privacy issues in apps that pose risks to users *unintentionally* due to vulnerabilities. Whereas in the second part, we measure and mitigate security issues in apps that are designed to *intentionally* put users at risk. Overall, this thesis aims to address the following research questions:

RQ1 Are vulnerabilities in SSL implementations that enable successful man-in-the-middle attacks of network connections prevalent in Android apps? If so, can we provide better recommendations that help address these problems?

RQ2 Do apps that claim to provide security and privacy properties that protect user information actually do?

RQ3 Can we design new malware detection tools that are more robust to malware and Android framework evolution than the state-of-the-art?

RQ4 Does having humans test apps during dynamic analysis improve malware detection compared to pseudorandom input generators such as Monkey [107]?

RQ5 How do different analysis methods (i.e., static, dynamic, and hybrid analysis) compare

to each other in terms of malware detection performance when the same technique is used to build the detection models?

To address these research questions, we carry out a set of experiments which we summarize following, and describe in more details in Chapter 4 – 7.

With respect to RQ1, we statically and dynamically investigate the SSL implementations of popular apps (i.e., apps that have been downloaded at least 10 million times on Play Store). During dynamic analysis, we define three attack scenarios that allow us to test for different vulnerabilities (Chapter 4). To address RQ2, we investigate apps that claim to provide three specific security and privacy properties (that is, anonymity, end-to-end encryption, and ephemerality). We then compare our observation of the operations of these apps in the presence of an adversary that aims to violate these properties, to the expected behaviors when the security and privacy properties are implemented properly (Chapter 5).

With respect to RQ3, we design and implement a system called MaMaDroid, which employs a novel approach in the detection of malware by abstracting API calls to fine-grained levels and then, modeling the behavior of the apps from the sequence of abstracted calls as Markov chains. We evaluate it using datasets spanning several years and compare it to a state-of-the-art detection system (Chapter 6). To address RQ4, we modify a virtual device that is designed to crowdsource human inputs for testing Android apps by integrating the detection technique used by MaMaDroid into it. We then instantiate the modified virtual device by using both pseudorandom (Monkey) and user-generated inputs. Finally, to address RQ5, we evaluate each analysis method (i.e., static, dynamic, and hybrid analysis), using the same modeling approach (i.e., a behavioral model relying on Markov chains built from the sequences of abstracted API calls), the same features, and the same machine learning classifier (Chapter 7).

1.2 Summary of Contributions

This thesis investigates the security and privacy issues in Android apps that are consequences of vulnerabilities and malicious intent. In measuring and mitigating these issues, we make the following contributions.

1. We show that despite increased awareness of SSL vulnerabilities, many popular apps still have these vulnerabilities; as a result, they leak users' private information. We provide recommendations and tested code sample that can help address the problem of poor SSL implementation by Android app developers and finally, we investigate why static and dynamic analysis yield slightly different results.

2. Our work shows that ephemeral messaging apps, i.e., apps with "disappearing" messages are not always ephemeral as we can recover "expired" messages. Similarly, anonymous apps can identify their users across multiple sessions (even after uninstalling and re-installing the apps), while also not making it clear to whom a user is anonymous.

3. We design and implement MaMaDroid, an Android malware detection tool that employs a novel approach in the detection of malware by abstracting API calls to their class, package, or family, and model the behavior of the apps through the sequences of API calls as Markov chains. We show that MaMaDroid can detect unknown malware samples from the same year as the training set with an F-measure of 0.99, with the detection accuracy still considerably high years after (i.e., 0.75 after two years), implying that MaMaDroid is robust to evolution in malware development and changes in the Android API. We also demonstrate the effect of abstraction to MaMaDroid's detection performance by abstracting API calls and using frequency analysis technique to model apps, showing that our approach performs better than other frequency analysis techniques without abstraction like DroidApiMiner.

4. We perform a comparative analysis of the different analytical methods i.e., static, dynamic, and hybrid in malware detection, showing that hybrid analysis performs best and that static analysis is at least as effective as dynamic analysis depending on how an app's input is generated during dynamic analysis. To perform malware detection that employs the same technique as MaMaDroid, we implement AuntieDroid, a virtual device that extends Chimp and allows for the collection of the method traces (from which features are extracted) produced by an app when executed. We find that pseudorandom input generator outperforms human users largely due to better code coverage. Our work also shows that dynamic code loading is prevalent in the wild, which could in theory be used by malware developers to evade static analysis tools.

1.3 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 provides background information on Android app components and the program analysis and machine learning algorithms applied in this thesis. Chapter 3 reviews prior work on the security and privacy of the Android ecosystem. Chapter 4 and 5 present work that measures vulnerabilities present in apps that, when exploited by malevolent actors, would lead to security and privacy issues. Chapter 6 and 7 discuss our efforts to mitigate security and privacy issues resulting from maliciously crafted apps. In Chapter 8, we conclude and discuss possible future works.

1.4 Publications

The works that comprise this thesis have been published or are currently under submission at various international journal and conference venues and have been done in collaboration with other coauthors. This section reports where the work in each chapter have been published and my contributions towards their completion.

- The work presented in Chapter 4 has been published and presented at the 2015 ACM

Conference on Security & Privacy in Wireless and Mobile Networks (WiSec) [157] in collaboration with Dr. Emiliano De Cristofaro. My contributions in this study include: collecting the data, designing and implementing the methodology, analyzing the results as well as writing a draft of the paper.

- The work presented in Chapter 5 has been published and presented at the 2016 NDSS Workshop on Understanding & Enhancing Online Privacy [158] and as a poster at the 2016 ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec) [159] in collaboration with Dr. Emiliano De Cristofaro. My contributions in this study include: collecting the data, designing and implementing the methodology, analyzing the results, and the write up.

- The work presented in Chapter 6 is currently under submission after revision at the ACM Transactions on Privacy and Security (TOPS) [160] while a shorter version has been published and presented at the 2017 Annual Symposium on Network and Distributed System Security (NDSS) [144] in collaboration with Enrico Mariconti, Dr. Panagiotis Andriotis, Dr. Emiliano De Cristofaro, Dr. Gordon Ross, and Dr. Gianluca Stringhini. My contributions in this study include: collecting the data (the benign sample used in our evaluation); implementing the module of MaMaDroid that performs call graph extraction and call graph parsing into sequences; re-implementing the API call abstraction (i.e., as presented in the extended version [160] and in this thesis) module; implementing API call abstraction to classes in MaMaDroid and evaluating it; implementing and evaluating the variant of MaMaDroid i.e., FAM; re-implementing DroidApiMiner from the code provided by its authors as well as evaluating it; investigating the detection performance of MaMaDroid in comparison to FAM; investigating the detection performance of FAM in comparison to DroidApiMiner; investigating the reasons for the false positives and false negatives exhibited by MaMaDroid; analyzing the results of the evaluation I performed; and

writing the sections of the paper that describe the tasks I performed.

- The work presented in Chapter 7 has been published and presented at the 2018 Annual Conference on Privacy, Security, and Trust (PST) [156] in collaboration with Mario Almeida, Enrico Mariconti, Dr. Jeremy Blackburn, Dr. Gianluca Stringhini, and Dr. Emiliano De Cristofaro. My contributions in this study include: collecting the data (i.e., the dataset we use for evaluation) and data preprocessing; evaluating the static analysis using MaMaDroid's code; implementing the trace parsing, feature extraction, and classification in AuntieDroid; evaluating AuntieDroid; implementing and evaluating the hybrid analysis version of MaMaDroid; performing a comparative analysis of the detection methods w.r.t. detection performance and misclassifications within each analysis method; analyzing the results; and writing the sections of the paper that describe the tasks I performed.

Chapter 2

Background

In this chapter, we present background information of the key components of an Android app, program analysis, SSL/TLS protocol, and machine learning; which are the building blocks of the work in this thesis.

2.1 Android App Components

2.1.1 The Primary App Components

Android app components are the building blocks that comprise an Android app and they serve as the entry points via which a user or the Android OS can interact with an app. There are four types of app components, namely:

Activity. An activity is a user interface that serves as the entry point for a user into an app. That is, the visual information an app wishes to display to a user is contained in an activity of the app. Similarly, if an app wishes to collect user input, the forms are placed in an activity and displayed to a user.

Service. A service is an app component that runs in the background and is used to perform long-running tasks or tasks for a remote process. For example, a service can be used to sync data in an app with a remote server (e.g., syncing emails by an email app) without disrupting the use of the app or to play music in the background while another app is in the foreground. A service does not provide an interface via which users can interact with it.

Broadcast Receiver. A broadcast receiver is an app component that allows an app respond

to system or app-defined event announcements. Because a broadcast receiver can be used to respond to system-wide events, it allows an app define operational flows different from user flows. That is, an app can perform a sequence of operations without user intervention. For example, an email app can be designed to start a service that starts syncing emails whenever it receives a broadcast that the device is connected to the Internet. Similarly, apps can also initiate broadcasts; for example, a contact app can let other apps *know* via a broadcast when a new contact is added. Because broadcast receivers are "another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running."[1]

Content Provider. A content provider is an app component that is used to manage data that is either private to an app or that is shared with other apps. Typically, the content/data is stored in a persistent storage location such as the file system, SQLite database, web, or shared preferences. Multiple apps can query or modify the data on a content provider if it permits it and the apps have permission to do so. For example, Android provides a content provider that manages the contact information in an address book, allowing apps with permission to access it, to query the content provider, e.g., to read contact details about a particular contact.

2.1.2 Other App Components and Definitions

Manifest. The Manifest is a file that declares all the components (i.e., activity, service, broadcast receiver, and content provider), that comprises an Android app. It also contains information about permissions, minimum API level, external libraries, and hardware features that the app requires to function correctly.

Android Package (apk). apk is an Android archive file format that contains the compiled source code, manifest, resources, assets, and libraries of an app.

[1] https://developer.android.com/guide/components/fundamentals

API Call. It is a request made to an Application Program Interface (API) to perform a task. In the Android *appified world*, an API call is a request to the Android framework.

Permission. Permissions are access control mechanisms defined and enforced by Android to protect resources and assets such as: files, sensors, hardware properties etc. An app will *only* have access to these protected assets if they have been granted permission to them by a user. Because access to assets are done via API calls, permissions are mapped to API calls i.e., the OS will only execute certain API calls *if and only if* the app has the necessary permission.

2.2 SSL/TLS Protocol

The SSL/TLS (written as SSL from now on) protocol is a security protocol that is used to encrypt data between two communicating parties. This section presents an introduction of the handshake protocol that begins the setting up of SSL between two parties, man-in-the-middle (MiTM) attacks that target parties employing SSL, and how the protocol is implemented by Android app developers.

SSL Handshake. To establish a secure connection between two parties – denoted as client (or app) and server – the client initiates the so-called SSL handshake, beginning with *client hello*, as depicted in Figure 2.1. Next, the client verifies the identity of the server: if the verification is not implemented correctly, an attacker can impersonate the server, thus leading to information leakage. Verification involves checking the chain-of-trust in server's certificate [62], hostname [116], and certificate validity [62,179]. It may also include verifying server's certificate serial number, key usage etc., or any developer-specific requirements. Failure in the verification should lead to termination of the connection.

SSL MiTM Attacks. In a MiTM attack, an attacker *sits* between two communicating parties and intercepts, read, modify, and/or relay messages between the parties, making them believe that they are instead talking directly to each other. In a MiTM attack

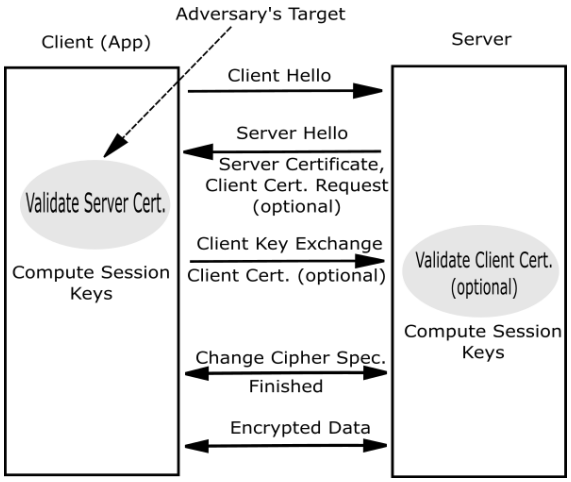


Figure 2.1: SSL Handshake.

over SSL, an attacker must subvert the certificate verification in the SSL handshake to be successful. The attacker does so aiming to, e.g., read and/or modify encrypted traffic between the client and server. The attacker succeeds if one of the following holds:

- The client accepts all certificates as the server's;
- The client does not verify the identity of anyone claiming to be the server;
- The server's certificate is forged by the attacker;

SSL Implementation in Android. To provide security between apps and servers, app developers turn to SSL over HTTP (HTTPS) sockets. Specifically, they rely on the android.net, javax.net.ssl, and java.security.cert packages which can be used to create secure sockets, administer, and verify certificates and keys. java.security.cert provides classes and interfaces for verifying the revocation status of a certificate. The javax.net.ssl package provides developers with `TrustManager` and `HostnameVerifier` interfaces, which are used in the SSL certificate validation logic. `TrustManager` manages all Certificate Authority (CA) certificates used in assessing a certificate's chain-of-trust validity. Only root CAs trusted by Android are contained in the default `TrustManager`. The `HostnameVerifier`

interface performs hostname verification whenever a URL's hostname does not match the hostname in the peer's identification credentials.

Developers can use the `TrustManager` and `HostnameVerifier` provided by the Android SDK as is or customize it to fit their need. Customization usually involves using developer-specified credentials, validation logic on `TrustManager` and/or `HostnameVerifier` interface. Reasons for customization can include any of the following:

- Certificate issued by a CA unknown to the OS
- Use of self-signed certificate
- Pinning peer credential ("SSL pinning")
- Server configuration sending incomplete certificate chain
- Use of a single certificate for multiple hosts

SSL Pinning involves coupling a host's trusted credential (e.g., an X.509 certificate or a public key) to its identity. Any credential received other than the one coupled will not be accepted as valid for the host, whether or not the credential is issued by a valid CA or it is trusted by the OS. This is usually done when a developer knows beforehand the trusted credentials of the host.

2.3 Program Analysis

The work presented in this thesis has involved one or more forms of program analysis. Here, we first introduce program analysis and then the methods (i.e., with respect to execution of the program) we have employed in our work *in the context of analyzing Android apps.*

Program analysis offers static compile-time techniques for predicting safe and computable approximations to the sets of values or behaviors arising dynamically at runtime when executing a program [151]. That is, it is an analysis performed to determine the

correctness, robustness, and safety of a program. It is applied in program optimization – detecting and avoiding redundant/superfluous computations – and program validation – ensuring that a program does what it is supposed to do (correctness) or reducing the likelihood of malicious behavior (safety). In order for an analysis to be computable, the result of the analysis is usually reached from an approximation of a program behavior. The analysis can be performed statically, dynamically, or a combination of both.

Static Analysis. This is a program analysis method that automatically analyzes a program (in this work, apps) without executing it by examining the source or byte code. The pros of using this analysis are that: it analyzes all portions of the source code and depending on the constraints (e.g., context or path sensitivity), it can be scalable and fast. At the same time, it has the following cons: it overestimates the code i.e., some portions of code it analyzes may never be executed during runtime and it cannot analyze any code that is loaded during runtime.

Dynamic Analysis. In this program analysis method, a program must be executed for it to be analyzed. This approach has the following pros: it is able to examine portions of code that are dependent on the runtime environment; it is able to examine any code that is loaded during execution; and it examines portions of code that are reachable following any number of execution paths. The cons of this approach include: it may be limited due to how inputs to the app are generated during test; it only examines portions of code that are executed during runtime; and the environment may influence the behavior of the examined app.

Hybrid Analysis. Hybrid analysis is program analysis that combines both static and dynamic analysis in the examination of an app. It inherits the pros of both static and dynamic analysis and some of the cons inherent in one approach that the other does not address.

There are four main approaches to program analysis, which include: abstract interpretation, control flow analysis, data flow analysis, and type and effect systems [151]. Abstract

interpretation is a technique used to derive an *abstract semantics* from which information can be obtained about a program (*concrete semantics*) i.e., it is used to approximate a program [63]. While control flow analysis is a technique that is used to determine the control flow or the execution order of statements or instructions in a program. Data flow analysis is a technique used to analyze the flow of data in a program in order to determine their values at each point of the program and how they change over time. While type and effect systems are an approach for augmenting static type systems to reason about and control a program's computational effects [145].

2.4 Machine Learning

In this section, we introduce machine learning (with a focus on classification) and the machine learning algorithms we employ to solve the classification problems our work addresses.

Machine Learning (ML) is an inductive process that involves automatically building a *classifier* for classifying or clustering objects into categories or classes by observing the characteristics (i.e., features) of a set of objects [183]. This process can either be *supervised* or *unsupervised*. In the former, a set of objects (*aka* ground truth) are manually classified into classes $C_{1...n}$ by a domain expert using a set of observed features represented as vector, and the goal is for a classifier to automatically *learn* from these features, and be able to predict the classes in $C_{1...n}$ of new unseen objects using their features. Whereas in the latter, the goal is for a classifier to cluster objects using their features, into different clusters, each of which can be assigned a class and meaning. In this thesis, we employ supervised ML as the goal of our experiments (in Chapters 6 and 7) is to classify an app into one of two classes: benign or malicious.

Training, Validation, and Test Sets. Typically, in supervised learning, the objects used to train and assess models are divided into three: training, validation, and test sets. The training, validation, and test sets are used to, respectively, fit the models, estimate pre-

diction error for model selection, and assess the generalization error of the final chosen model [93]. Validation of models can also be performed using the k-fold cross validation approach, in which k different models are built by partitioning the initial objects into approximately equal-sized k sets: $O_1, O_2, ..., O_k$, and then iteratively applying a train-and-test approach on the partitions using a different partition as the test set in each iteration. The final evaluation metric values of the model is obtained by computing the average of the effectiveness of each individual model.

The ML algorithm used to fit models in the training phase depends on several factors that include: the type of problem, the size of the objects' features in relation to the size of the total amount of objects, number of classes of the objects, time to train the models etc. There are several ML algorithms used for solving classification-related problems; here, we only introduce the three algorithms we use in our work. We refer readers seeking to know more about ML and ML algorithms to prior works [34, 93, 141, 183].

K-Nearest Neighbor (KNN). KNN [92] is an ML algorithm that is used for classification and regression. In solving a classification problem, it attempts to predict the class of a data point (object) based on the class of a predefined k number of training examples (objects) closest in distance to the data point. Generally, larger k values suppresses the effects of noise on the classifier, but makes the class boundaries of the data points less distinct.

Random Forests. Random Forests [36] is an ensemble ML algorithm that builds several independent tree-structured estimators (classifiers) from a subset of the training data and then returns the average of their predictions as the prediction of the classifier. The larger the number of estimators, the longer the time taken to train and classify samples but the better the classifier. Although, beyond a certain number of estimators, there exists no significant increase in the prediction results of the classifier.

Support Vector Machine (SVM). SVM [113] is an ML algorithm that is used for classification, regression and outliers detection. It operates by finding the hyperplane – a decision boundary that splits the input variable space by their class – that creates the biggest mar-

gin between the training data points of different classes [93]. The margin is calculated as the distance from the hyperplane to only the closest points and only these points, called the support vectors are relevant in defining the hyperplane and in the construction of the classifier [37]. SVM is very effective when processing objects with high dimensional spaces but may suffer from overfitting when the number of features of each object is much greater than the number of objects.

Classification Metrics. There are several evaluation metrics used to show the effectiveness of an ML classifier. Here, we only describe the metrics in the Scikit-learn Python library [162], which we use in our work.

Precision. Precision shows the ability of a classifier to predict correctly, the class of an object. It is given as the ratio $tp/(tp + fp)$ where tp is the number of true positives, i.e., positive objects correctly predicted as positive ones and fp the number of false positives, i.e., negative objects wrongly predicted as positive objects.

Recall. Recall shows the ability of a classifier to find all the positive [or relevant] objects. It is given as the ratio $tp/(tp + fn)$ where fn is the number of false negatives, i.e., positive objects wrongly predicted as negative ones.

F1-Score. The F1-Score also known as F-Measure or F-Score, is the harmonic mean or weighted average of the precision and recall of a classifier. It is given as:

$$F1 = \frac{2 * (Precision * Recall)}{(Precision + Recall)}$$

Other metrics in the Scikit-learn Python library [162] that can be used to evaluate a classifier include: Accuracy, Area Under the Curve (AUC), Receiver operating characteristic (ROC) curve, Brier score, Cohen's kappa etc.

Chapter 3

Literature Review

In this chapter, we review prior work on the security and privacy of the Android OS, Android apps, and app market.

Over the years, as Android's popularity increased, the research community has investigated its security, usability, users' security and privacy, Android apps etc. In this chapter, we review studies that investigate: (i) the security and privacy of the Android OS, focusing on vulnerabilities discovered as well as API abuse; (ii) apps that pose security and privacy risk either because of the vulnerabilities they contain or because they are malicious; and (iii) the markets that distribute these apps, focusing on the characteristics of the apps that comprise the markets and the measures if any, employed to vet the apps.

3.1 Security and Privacy in the Android OS

3.1.1 Vulnerabilities

We report here, prior work that examined the security of the Android OS, reporting vulnerabilities that allow for malicious activities.

Privilege Escalation. One of the approaches used in Android to provide security is the implementation of access control and sandboxing. That is, system resources and components as well as inter-process communications are protected by permissions that an app must possess for it to access them. Consequently, each app is assigned a *UserId* which is mapped to a set of permissions granted to the app at installation or run time and can be modified[1]

[1]Prior to Android version 6.0, the permission is fixed at installation time and cannot be modified afterwards by the app or the user.

at any time by the user. During runtime, a reference monitor checks whether access to protected resources or inter-component communication should be permitted. Several studies [39,68,71,87] have investigated this access control mechanism in Android and show that it is vulnerable to the confused deputy attacks [110], allowing apps to perform operations or access resources they do not have permission for. Even though Bugiel et al. [39] propose possible solutions to this problem, these attacks are still possible [114] four years after, highlighting the importance of continuous research work and development of deployable solutions to known Android security and privacy problems.

Device Rooting. Device rooting is another form of privilege escalation that exploits vulnerabilities (such as mishandling copy-on-write[2], `setuid` resource exhaustion failure[3]) in the Android kernel, allowing apps execute arbitrary code in the kernel rather than exploiting the confused deputy problem. Apps that exploit these vulnerabilities pose a serious security and privacy risk, as they can execute any command with elevated privilege, mask their activities, and are hard to remove. Several studies [109,115,226] have discovered apps packed with root exploits, which have consequently led to the development of techniques for detecting rooted devices or apps containing root exploits [80,115]. Recently, Suarez-Tangil and Stringhini [191] reported the increased use of native code by malicious apps which often contain vulnerability exploits; one of which is used to root a device. This corroborates the findings of Ho et al. [115] that known malicious apps that employ root exploits, all do so via third-party native code.

Memory Corruption. One method via which memory corruption occurs is via buffer/heap overflow. A number of research efforts [72,112,186] have discovered vulnerabilities in the Android framework that allows a number of possible attacks, such as remote code execution, data exfiltration, and privilege escalation. For instance, vulnerabilities in the Stagefright media framework of Android was exploited using integer overflow and allowed an attacker

[2] `https://access.redhat.com/security/vulnerabilities/DirtyCow`
[3] `https://nvd.nist.gov/vuln/detail/CVE-2008-0008`

remotely execute code on the device [72].

Other vulnerabilities that have been reported in previous studies include: data residue vulnerability – a vulnerability that results from uncleared data (e.g., files, database, capabilities, etc.) of an uninstalled app being inherited by a new app, thus resulting in privacy leaks or privilege escalation [221]; side channel leakages that allow for runtime information gathering attacks [220].

A comprehensive list of security vulnerabilities that have been discovered on Android are available in the CVE database [12] and the vulnerabilities that have been patched by Google are available in the Android Security Bulletin [11]. Usually, these vulnerabilities are exploited by malicious actors via two mechanisms: drive-by download and malicious apps [147]. This thesis focuses on vulnerability measurement and mitigation on apps since they form a primary mechanism for vulnerability exploitation. A review of studies that have investigated vulnerabilities in potentially benign apps as well as malicious apps designed to exploit these vulnerabilities is presented in Section 3.2.

3.1.2 Framework API Abuse

Here, we discuss API calls that are intended by Google to allow developers perform certain operations but have been used to carry out malicious activities. These API calls may or may not have been deprecated or modified.

Permissions Abuse. Several research studies [21, 28, 74, 85, 87, 182] have investigated the use of permission by app developers, with Au et al. [27] mapping permissions to their corresponding API calls. While permissions serve as an access control mechanism to protect access to device and user information, they have been overly abused by developers requesting many permissions they often do not need [41]. These overly-permissioned apps pose privacy risks to users, especially when they have vulnerabilities malicious apps can exploit; thus, they serve as side-channels for leaking data. Also, the Android framework allows developers define custom permissions which they can use to regulate access to their app

components by other apps. Work by Sellwood and Crampton [184] and more recently by Tuncay et al. [201], show how these can be abused to perform privilege escalation attacks and unauthorized access to components. Finally, Sadeghi [178] presents a number of tools for detecting and preventing security issues that are permission-induced.

GUI Confusion. The Android framework provides API calls for drawing views and activities as well as allowing a developer choose where to position them. Prior works [32,89,138, 152] show that these API calls can be used maliciously in GUI attacks that can result in tapjacking, clickjacking, privacy leaks etc. Specifically, they show that an app can draw on top of other apps using the `addView`, `startActivity`, `moveTaskToFront`, `moveTaskToBack`, and the setView and show (for Toasts) methods. Depending on the flags set in these methods, an app can use them to intercept user inputs by either switching the UI the user intends to interact with or allow the inputs pass through from the intended UI. Bianchi et al. [32] developed a tool that detects and mitigate these GUI attacks and informs users of an app whether the GUI they are interacting with belongs to the app or another app. Recently, Fernandes et al. [89], Ren et al. [176], and Kraunelis et al. [131] show that these defense mechanisms are not sufficient and that GUI attacks are still possible.

Dynamic Code Loading. The Android framework provides developers with API calls that allow them dynamically load and execute code either from resources or the Internet. Previous studies [142, 167, 222] show that these API calls are used by malware to load malicious code or as an evasion technique to prevent detection. For example, before apps are made available for download on Play Store, they are first examined for maliciousness and other characteristics that Google has defined in its vetting process [136,199]. To defeat this vetting process malicious developers can submit their apps without the malicious blocks of code and dynamically load these blocks of code when the apps are running on users' devices.

3.2 Apps: A Driver for Security and Privacy Loss

While the Android OS may provide measures that enhance users' security and privacy, apps that run on the OS may yet expose users to security and privacy risks either intentionally (i.e., malware) or unintentionally (i.e., as a result of vulnerabilities). In this section, we review studies that examine security and privacy loss via Android apps.

3.2.1 Vulnerabilities

Here, we review prior works that investigate vulnerabilities in apps that could result in the loss of users' security and privacy when exploited by malevolent actors.

Privacy Leaks. In late 2013, researchers from Gibson Security discovered a flaw in Snapchat's API that allows an adversary to reconstruct Snapchat's user base (including names, aliases, phone numbers) within one day as well as the mass creation of bogus accounts [208]. Ikram et al. [121] investigated the security and privacy risks of apps with VPN permission, finding that 18% of the apps they analyzed implement tunneling protocols without encryption, thereby exposing users to security and privacy loss. As a result of some apps transmitting user information in plain text or accessing and transmitting sensitive information without the knowledge of users, a number of research efforts have designed tools that track sensitive information flow on a device. In particular, TaintDroid [76] taints the sources (e.g., device sensors) of sensitive information and tracks their flow whether they lead to sinks (e.g., network socket) whenever an app accessed them. AppAudit [215] employs hybrid analysis to detect information leakage at a true positive rate of 99.3% and no false positives.

SSL/TLS Implementation Vulnerabilities. Another means of privacy leakage in Android apps is via vulnerable SSL implementation. This usually involves implementing vulnerable `TrustManagers` and `HostnameVerifiers` that allow successful MiTM attacks (see Section 2.2). Several studies show that apps [82, 95, 189] and the underlying libraries [38, 101] used by apps contain vulnerabilities that result in the leakage of user information. Fahl et

al. [82] employ static analysis to discover vulnerabilities in Android apps. They complement their analysis with a manual dynamic analysis of the apps to confirm their results. While they attempt to discover vulnerabilities via static and dynamic analysis, their work does not investigate whether these analysis types yield differing results and why. Conti et al. [61] and Sounthiraraj et al. [189] introduce tools (respectively, MITHYSApp and SMV-Hunter) for automated SSL vulnerability discovery. While MITHYSApp is designed to run on users' device, SMV-Hunter employs static analysis for large-scale automated SSL MiTM Vulnerability (SMV) discovery. Kranch and Bonneau [130] perform a measurement study of the adoption and understanding of HTTP Strict Transport Security (HSTS) and public key pinning. Their study reveals a lack of understanding of these security features and that cookies can be leaked from domains that implement key pinning to malicious scripts in HTTP domains they load resources from.

Georgiev et al. [101] studied the libraries and APIs used in SSL implementation by apps, finding that they are broken largely due to bad API design that leads to developers misinterpreting and misunderstanding parameters and return values. Similar to [101], Brubaker et al. [38] implemented a means for large-scale adversarial testing of certificate validation logic in different SSL libraries. Their study reveals that there are validation discrepancies between popular SSL libraries such as OpenSSL and NSS, where one library accepts a certificate that another rejects. Bates et al. [29] introduce CERTSHIM, a tool for certificate verification retrofitting that is dynamically hooked to SSL and data transport libraries to prevent SSL vulnerabilities. For a survey of other proposed solutions to detecting and fixing the SSL vulnerabilities, and general SSL usage in Android apps, readers can see [210] and [174], respectively.

Although, these works have highlighted important vulnerabilities in SSL implementations by Android app developers, it is interesting to evaluate after several years, whether these vulnerabilities are still prevalent especially in very popular apps with millions of downloads. As reported by Fahl et al. [83], developers with vulnerable apps make their

apps vulnerable usually because they want to use self-signed certificate as against buying one and also to circumvent errors – copying SSL code from platforms such as Stack Overflow.[4] This is confirmed by Fischer et al. [91] who recently showed that 183k apps contain insecure TLS code copied from Stack Overflow. Therefore, providing developers code that do not result in security errors and/or safe use of self-signed certificate might help mitigate these vulnerabilities.

3.2.2 Malware

Here, we review prior works that detect apps developed to act maliciously. The research community has developed a number of ways to examine these apps, with each approach having pros and cons. The approaches generally used are static, dynamic, and hybrid analysis. While static analysis allows for the overestimation of the code in an app, it does not capture runtime behavior of apps. Although dynamic analysis captures runtime behavior of the apps under analysis (AUA), it does not analyze behavior of the AUA not observed during runtime as a result, may falsely classify the behavior of the AUA. To obtain the best of static and dynamic analysis, they are combined in a hybrid manner to examine AUA. Below, we review studies that have employed these approaches to detect malicious apps.

3.2.2.1 Static Analysis

Android malware detection based on static analysis aims to classify an app as malicious or benign by relying on features such as permission request, usage or sequence of API calls, *Intent*, etc. extracted from the app's apk.

Permissions. Prior works [77, 118, 180, 182] have introduced techniques that build features from the *permissions* requested by apps, leveraging the fact that malware often tend to request dangerous or unneeded permissions. However, this approach may be prone to false

[4] `https://stackoverflow.com/`

positives as benign apps may also request dangerous permissions [77]. Moreover, since Android 6.0, the permission model allows users to grant permissions at runtime when they are required for the first time, thus some dangerous permissions might never actually be granted (in fact, app developers often request permissions that are never used [85]).

API Calls. Prior work has also relied on the usage of specific API calls or the sequence of calls for malware detection. DroidAPIMiner [14] performs classification based on the API calls that are more frequently used by malware. It relies on the top 169 API calls that are used more frequently in their malware set than in their benign set, along with data flow analysis on calls that are frequent in both benign and malicious apps, but occur up to 6% more in the latter. TriFlow [148] ranks apps based on potential risks by using the observed and possible information flows in the apps. DroidMiner [217] studies the program logic of sensitive Android/Java framework API functions and resources, and detects malicious behavior patterns. Hou et al. introduce HinDroid [117], which represents apps and API calls as a structured heterogeneous information network, and aggregates the similarities among apps using multi-kernel learning. Finally, ASTROID [88] uses common subgraphs among samples as signatures for identifying unknown malware samples, while RevealDroid [97] employs supervised learning and obfuscation-resilient methods targeting API usage and intent actions to identify their families.

However, due to changes in the Android API as well as the evolution of malware, systems based on the usage of specific API calls require frequent retraining as new API versions are released and new types of malware are developed. Deprecation and/or addition of API calls with new API releases is quite common, and this might prompt malware developers to switch to different API calls. Also, systems that model the API calls as signatures can be evaded by polymorphism and obfuscation, or by call re-ordering attacks [129].

Manifest and Byte Code. The manifest of an app has also been used in the detection of malware as features such as permissions, intent filters etc. can be used to distinguish between malicious and benign apps. Droidmat [214] uses API call tracing and manifest

files to learn features for malware detection. Teufl et al. [200] apply knowledge discovery processes and lean statistical methods on app metadata extracted from the app market, while Gascon et al. [100] rely on embedded call graphs. MAST [47] statically analyzes apps using features such as permissions, presence of native code, and intent filters and measures the correlation between multiple qualitative data. Similarly, RiskRanker [109] detects zero-day Android malware by assessing the potential security risks of apps. It examines an app to detect certain behaviors it classifies as `unsafe`, such as native code encryption and dynamic code loading which in conjunction with "attack code" that can exploit known vulnerabilities could pose risk to users. Canfora et al. [44] follow two approaches in the detection of malware: 1) a Hidden Markov model (HMM) to identify known malware families and 2) a structural entropy that compares the similarity between the byte distribution of the executable file of samples belonging to the same malware families.

DREBIN [25] learns malware and goodware patterns offline, with the trained model transferred to a device where it identifies malware by performing a broad static analysis on the apps on the device. This is achieved by gathering numerous features from the app's manifest (intents, hardware components, app components) as well as the app's byte code (restricted and suspicious API calls, network addresses, permissions). Malevolent behavior is reflected in patterns and combinations of extracted features from the static analysis. For instance, the existence of both SEND_SMS permission and the `android.hardware.telephony` component in an app might indicate an attempt to send premium SMS messages, and this combination can eventually constitute a detection pattern. Alas, techniques based on decompiled code can be evaded using dynamic code loading, reflection, and the use of native code [167, 173].

Machine Learning. Machine learning techniques have also been applied to assist Android malware detection. Chen et al. [53] proposed KUAFUDET that uses a two-phase learning process to enhance detection of malware that attempts to sabotage the training phase of machine learning classifiers. Similarly, Transcend [126] presents a framework for identifying

aging machine learning malware classification models, i.e., it uses statistical metrics to compare the samples used for training a model to new unseen samples to predict degradation in the detection accuracy of the model.

3.2.2.2 Dynamic Analysis

Dynamic analysis based techniques attempt to detect malware by capturing the runtime behavior of an app, targeting either generic malware behaviors or specific ones. DroidTrace [224] uses ptrace (a system call often used by debuggers to control processes) to monitor selected system calls that allow for the execution of dynamic payloads, and classify their behavior as, e.g., file access, network connection, inter-process communication, or privilege escalation. Canfora et al. [42] extract features which they then use to classify an app as malware or benign from the sequence of three system calls by executing the apps on a Nexus 5 smartphone, while Lageman et al. [132] model an app's behavior during execution on a VM using both system calls and `logcat` logs. CopperDroid [196] employs dynamic analysis to automatically reconstruct malware behavior by observing executed system calls. While CrowDroid [40], a lightweight client running on a device, captures system calls generated by apps and sends them to a central server, which builds a behavioral model of each app. Using strict sequences of system or API calls could be more easily evaded by malware, which may map the strict sequences as signatures and then add unnecessary calls to evade detection .

Finally, Chen et al. [56] propose a Markov chain-based model that dynamically analyze system and developer-defined actions from intent messages (used by app components to communicate with each other at runtime). They then probabilistically estimate whether an app is performing benign or malicious actions at run time but obtain low accuracy (an average of 0.67) overall.

Prior work typically perform dynamic analysis using automated input generators to exercise apps when the device is running. These generators exercise apps using random

fuzzing [140, 218], concolic testing [20, 102], or generating pseudorandom input [107]. However, this might be inadequate due to the low probability of triggering malicious behavior, and can be side-stepped by knowledgeable adversaries, as described by Wong and Lie [212]. In addition, dynamic analysis techniques can only detect malicious activities if the code exhibiting malicious behavior is actually triggered during the analysis.

3.2.2.3 Hybrid Analysis

A number of tools have combined static and dynamic analysis, e.g., by using the former to analyze an apk and the latter to determine what execution paths to traverse, or by combining features extracted using both static and dynamic analysis. Andrubis [134] is a malware analysis sandbox that extracts permissions, services, broadcast receivers, activities, package name, and SDK version from an app's manifest as well as the actual bytecode. It then dynamically builds a behavioral profile using static features as well as selected dynamic ones, such as reading/writing to files, sending SMS, making phone calls, using cryptographic operations, dynamically registering broadcast receivers, loading `dex` classes/native libraries, etc. Chen et al. [54] introduce StormDroid which extracts static features, such as permissions and API calls, and extend their features vector to add dynamic behavior-based features. While their experiments show that their solution outperforms, in terms of accuracy, other antivirus systems, the quality of their detection model critically depends on the availability of representative benign and malicious apps for training. MADAM [181] extracts five groups of features (system calls, sensitive API calls, SMS, user activity, and app metadata) at four different layers – kernel, application, user and package – which are used to build a behavioral model for apps, and uses two parallel classifiers to detect malware.

Marvin [133] uses features from both static and dynamic analysis to award malice scores (ranging from 0 to 10) to an app and classify as malware, apps with scores greater than 5, while CuriousDroid [46], an automated user interface (UI) interaction for Android apps, integrates Andrubis [134] as its dynamic module in order to detect malware. It decomposes

an app's UI on-the-fly and creates a context-based model, generating series of interactions that aim to emulate real human interaction. IntelliDroid [212] introduces a targeted input generator that integrates with TaintDroid [76] aiming to track sensitive information flow from a source (e.g., a content provider such as contact list database) to a sink (e.g., network socket). It allows the dynamic analysis tool to specify APIs to target and generates inputs in a precise order that can be used to stimulate the Application Under Analysis (AUA) to observe potential malicious behavior.

Since there are several entry points into an Android app (e.g., via an activity, service, and broadcast), dynamically stimulating an AUA is usually done using tools like Monkey or MonkeyRunner, or humans. Targeted input generation tools such as CuriousDroid and Intellidroid aim to provide an alternative stimulation of apps that is closer to stimulation by humans and more intelligent than Monkey and MonkeyRunner. Finally, we refer the reader seeking more details on the large body of work on Android malware to prior works on useful surveys of Android malware families and detection tools [18, 84, 195], as well as an assessment of Android analysis techniques [175].

3.3 Security and Privacy in Android App Stores

Due to app stores serving as a repository and channel for app distribution, they pose serious security and privacy risks to users; especially, when they act maliciously or are hacked by malicious actors. Thus, related work have evaluated app stores using apps on the store and user reviews to reach conclusion about the security or trust levels of the store.

3.3.1 Apps on the Store

Jansen and Bloemendal [123] define an app store as: "An online curated marketplace that allows developers to sell and distribute their products to actors within one or more multi-sided software platform ecosystems". Their conceptual model of an app store highlights

that an app store should have policies that regulate functional features such as app quality. Therefore, we review here, prior works that have reached conclusion of app stores based on the apps available on the store.

As a result of the limited accessibility of Play Store in China, most Android users in China install apps from other third party app store. In order to evaluate the trustworthiness of some of these app stores, Ng et al. [150] measure the similarity and difference in content of the apps on these stores to the official apps available on the developers' website. They find that 37.74% of the apps they obtain from these app stores, are either an older version of the official app or have resources that have been modified. Similarly, they find that the permissions or the code have been modified in 36.17% of the apps. Due to the nature of these modifications (permissions, resources, and code) the resulting apps on the app store could be acting maliciously, as a result, the app stores are awarded a low trustworthy ranking score (47.37 over 100). Kikuchi et al. [128] also investigated the mitigation efforts five app stores (including Google's Play Store) apply in preventing the upload of malicious apps to their stores. They evaluate the mitigation efforts using three different metrics: malware presence ratio, malware download ratio, and malware survival period. They find that malware presence in three third-party markets is about ten times higher than in Play Store and that only Play Store takes active steps to remove malware.

While Ng et al. [150] reveal apps that have been modified on third party stores, other studies [50, 64, 225] investigate the distribution of repackaged apps in different app stores. With Crussell et al. [64] revealing that at least, 141 apps have been cloned multiple times, some as many as seven times, and Chen et al. [50] showing that repackaging is used by malevolent actors to include advertisements or malicious code in legitimate apps.

Finally, a number of other research studies [51, 104, 137, 223] have also proposed ways for app stores to verify or vet apps they distribute. Fahl et al. [81] propose a framework – Application Transparency (AT) – for defending against very powerful adversaries and DroidSearch [172] presents pre-filtering techniques that allow for quick retrieval of candidate

apps from app stores that should be further analyzed for maliciousness.

3.3.2 User Reviews

User reviews on apps usually indicate satisfaction or dissatisfaction with the apps, here, we review research efforts that have presented techniques for extracting knowledge from reviews.

Several studies [52, 94, 127, 161, 204] have developed data mining techniques with sentiment and topic analysis to extract useful information from online user reviews. Specifically, Iacob and Harrison [119] proposed MARA (Mobile App Review Analyzer) for automatically retrieving feature requests for app reviews on online markets. Maalej and Nabil [139] extended this work by applying string matching, sentiment analysis, Natural Language Processing (NLP), and metadata review to user reviews from the Apple App Store and Google Play Store for automatic classification of the reviews as bug reports, feature requests, user experiences, and ratings. They achieve precision and recall for all four classes that ranges from 71% to 97%. Similar to their work is [161], that also uses NLP, text and sentiment analysis to automatically classify reviews into defined categories like maintenance and evolution. Although many of these studies present tools and frameworks for extracting information about feature requests and bugs, security and privacy researchers can employ similar techniques in selecting apps for vulnerability analysis.

Concluding Remarks. From our review of previous literature, we observe some interesting patterns and gaps. First, over time, the same or similar vulnerabilities are discovered in the Android OS or apps. For example, Android framework API abuse that results in GUI attacks were reported in 2012 [138] and six years later, these attacks are still possible [131]. With respect to the OS, once a vulnerability is discovered, they are patched in the form of over-the-air updates pushed to devices in timely fashion depending on the severity of the

vulnerability. Similarly, the framework are also updated with new API version releases that deprecate vulnerable API calls and add new API calls. Whereas with respect to apps, it is not often clear whether developers fix vulnerabilities in their apps that are serious security and privacy threats to users or if they do so in a timely manner.

Second, static analysis malware detection tools based on API calls are likely easy to evade if malware authors learn what API calls are used as signatures for detection. For example, they could use a combination of different API calls that allow them achieve the same function. Moreover, the Android framework is constantly changing with the addition and deprecation of API calls with new API releases. Third, employing pseudorandom input generators to stimulate apps as used by dynamic analysis-based tools might be inadequate due to the low probability of triggering malicious behavior. That is, compared to a human user, pseudorandom input generators might only trigger events that malware are not interested in capturing due to the events not processing or accessing sensitive information.

This thesis thus, presents research methods and analysis we carry out to address these observations. We divide the thesis into two parts, with the first part (Chapter 4 and 5) focusing on vulnerability discovery and mitigation on Android apps (first observation). While the second part (Chapter 6 and 7) focuses on the detection of malicious apps (second and third observations).

Chapter 4

Experimenting with SSL/TLS Vulnerabilities in Android Apps

In this chapter, we address **RQ1**: Are vulnerabilities in SSL implementations that enable successful man-in-the-middle attacks of network connections prevalent in Android apps? If so, can we provide better recommendations that help address these problem?

The work presented in this chapter has been published in the 2015 ACM WiSec [157]

4.1 Motivation

Over the past few years, the number of always-on, always-connected smartphones has skyrocketed [98,99]. In 2013, 73% of mobile phone users regularly accessed the Internet via their mobile devices [13,190] and this ratio is likely to increase as sales of smartphones and tablets keep growing. Naturally, as the number of smartphones increase, so do the total amount of personal and sensitive information they transmit. Thus, it is crucial to secure traffic exchanged by these devices, especially considering that mobile users might connect to open Wi-Fi networks or even fake cell towers. The Transport Layer Security (TLS) protocol and its predecessor, Secure Sockets Layer (SSL) are cryptographic protocols that help to provide confidentiality, integrity, and authentication of network traffic. However, SSL/TLS implementations in smartphone applications (apps) have more bugs and are prone to vulnerabilities than browsers. For example, they are vulnerable to man-in-the-middle (MiTM) attacks [82,101]. Moreover, while browsers provide users with visual feedback that the communication is secured (via the lock symbol) and of certificate validation issues, apps

do so less extensively and effectively [19].

Prior works [38, 101] that studied the libraries and APIs used in SSL implementations by non-browser apps show that misunderstanding of the libraries have lead to developers designing apps that have enabled the leakage and retrieval of users' personally identifiable information (PII). Further, [61,82] show that vulnerabilities are also introduced for reasons such as the use of self-signed certificate or circumventing security errors. Although these studies [38, 189] have presented means for detecting vulnerabilities, it is unclear whether these are still prevalent in Android apps. Therefore, in this chapter, we address the research question, RQ1, introduced in Chapter 1.1, which we further divide into the following sub-questions:

Q1 Do popular apps contain SSL vulnerabilities that allow for the leakage of PII?

Q2 Do popular apps employ certificate pinning to enhance the security of their network traffic?

Q3 Can we provide useful recommendations that help developers that want to use self-signed certificate use them in a secure way?

The rest of this chapter is structured as follows: Section 4.2 introduces the methodology we employ to perform our experiments, while Section 4.3 presents the results. Finally, Section 4.4 discusses the implications of our results, the recommendations we make for the safe use of self-signed certificates, and the limitations of our work.

4.2 Methodology

In order to address these research questions, we perform static and dynamic analysis on 100 apps with at least 10M downloads (10% of all apps with at least 10M downloads, as per [2]), that request full network access. In this section, we discuss: our analysis methods;

the apps that comprise our dataset; the sensitive information we track for leakage; and the ethical considerations that we put in place.

4.2.1 Static Analysis

To carry out our analysis, we first perform a manual analysis of apps to understand the different ways developers introduce vulnerabilities in their apps. To do so, we use a custom script to decompile apps using dex2jar[1] and JD-GUI[2] and search for key terms such as `HttpsURLConnection`, `HostnameVerifier`, and `TrustManager`, which indicate the presence of SSL code. Then, we manually analyze the `TrustManager` and `HostnameVerifier` implementations of the apps to i) look for possible vulnerabilities and ii) check whether certificate pinning is employed.

TLSDroid. After manually analyzing the `TrustManager`s and `HostnameVerifier`s found in apps in our dataset, we then implement TLSDroid, which employs static analysis for large scale SSL vulnerability discovery. TLSDroid extends MalloDroid [82] and it is dependent on Androguard.[3] Unlike MalloDroid, it can correctly analyze multidex apps and it can tell when an app's SSL implementation is not vulnerable by checking for the invocation of methods used to validate a certificate. It works in two phases; in the first phase, it classifies an app as either vulnerable or not, based on whether it detects a vulnerable or correct SSL implementation, whereas in the second phase, it classifies those apps that have not been classified in the first phase into one of: vulnerable, potentially vulnerable, or not vulnerable. We describe these phases in more details below.

To detect vulnerable SSL implementations, we analyze the `TrustManager`s, `HostnameVerifier`s, and `WebView`s found in an app. As stated in Section 2.2, the `TrustManager` assesses the validity of a certificate's chain-of-trust, while the `HostnameVerifier` verifies that the hostname in the certificate matches the hostname of the connecting peer. To

[1] `https://code.google.com/p/dex2jar/`
[2] `http://jd.benow.ca/`
[3] `https://github.com/androguard/androguard`

do so, the Android framework API provides developers with a number of methods (e.g., `checkServerTrusted`, `checkValidity`, and `verify`) that can be used to check the validity of a peer's certificate. TLSDroid collects all the methods that `TrustManager`s and `HostnameVerifier`s use to validate chain-of-trust and hostname, respectively, as well as those that can be used to validate certificates in general. In its first phase, TLSDroid checks if the methods used to validate chain-of-trust or hostname are implemented and whether certificates are validated correctly. It reports an implementation as vulnerable if the method is empty, returns true or void without performing any checks, or instantiates an insecure `SSLSocketFactory` or `HostnameVerifier` i.e., `AllowAllHostnameVerifier`. With respect to the `WebView` (a view used to display web pages), it checks whether the implementation just returns true or void when an SSL error occurs i.e., the error is not handled and the connection may be established if possible.

If the methods used to validate chain-of-trust or hostname are implemented and they are not classified in the first phase, TLSDroid then moves to a second phase and collects all the API calls that are invoked in each method and performs a one step forward-flow analysis. It then iteratively checks for certificate validation in each of the collected API calls. If it does not find a correct certificate validation in the list of API calls checked, it reports the `TrustManager` or `HostnameVerifier` as potentially vulnerable. Whereas if it finds a correct implementation, it reports the app as not vulnerable and vulnerable otherwise. Here, we report an app as potentially vulnerable because it is possible that an n-step forward-flow analysis may not reveal a correct implementation. While we limit the step of the flow analysis to one to aid faster analysis (about 5 secs on average per app) in a large scale setting, the number of forward-flow analysis TLSDroid performs can be increased. In both phases, if the classes of all the vulnerable or potentially vulnerable methods are not referenced in any other class in the app, the app is not reported as containing vulnerability. We do this to reduce false positives that static analysis introduces due to code overestimation.

4.2.2 Dynamic Analysis

When analyzing apps dynamically, we manually exercise apps as this allows us use features of the apps that would request/need sensitive information and probe for vulnerabilities. In Chapter 7, we show how apps can be dynamically assessed in a large scale using a virtual device and allowing both humans and an automated pseudorandom input generator to exercise the apps. During our analysis, we assume the presence of an adversary that has control over the Wi-Fi access point a victim connects to and simulate the following possible MiTM attack scenarios:

S1: The adversary has their CA certificate with which they are able to generate valid certificates for any number of domains, installed on the victim's device;

S2: The adversary presents an invalid, self-signed certificate;

S3: The adversary presents a certificate with a wrong Common Name (CN) and/or SubjectAltName, signed by a valid CA.

S1 allows us address Q2 because apps that employ certificate pinning will reject connections to their servers because the [valid] certificates presented differ from the pinned certificates. S2 and S3 allow us address Q1 as apps that establish connections to their servers in these settings will leak user information. To address Q3, we encourage the use of certificate pinning and provide a code sample with a fail-safe default when the wrong certificate is used in the wrong environment i.e., development vs production.

Experimental Testbed. We conduct our experiments using an LG Nexus 4 smartphone running Android KitKat 4.4.4 that connects to a Lenovo laptop (running Windows 8.1), which acts as the Wi-Fi access point. To capture traffic and perform MiTM attacks, we install and run Wireshark[4] and Fiddler2[5] (we install Fiddler's certificate on the device as the trusted CA certificate used to sign other leaf certificates) on the access point. We

[4] `https://www.wireshark.org/`
[5] `http://www.telerik.com/fiddler`

conduct the experiments in the Autumn of 2014 and then reexamined them in March/April 2015. All statistics refer to the first round of the tests, however, whenever an app is no longer vulnerable (during second round of test), we report it in the text.

Sensitive Information. Throughout our experiments, we classify as sensitive, packets with the following information: login credentials, device information (e.g., IMEI number), location data, chat and email messages, financial information, appointments on calendar, contact list, and files.

Ethical Considerations. We password-protect (using WPA2) the access point in our experiments to ensure that only the test device can connect to it. Also, we conduct the experiments in a controlled test environment to ensure we *only* monitor or capture traffic from the network that we setup.

4.2.3 Datasets

To build a dataset of popular Android apps, we select 97 apps that request permission for full network access and that have been downloaded at least 10 million times (according to the official Android market – Play Store). We choose apps from several different categories (including social networks, gaming, IM, e-commerce, etc.) that access and process multiple kinds of sensitive information rather than those that only exchange login credentials. We also add 3 more apps that request full network access, even though they have less than 10M downloads: Barclays Mobile Banking, TextSecure, and Amazon Local. We choose Barclays Mobile Banking and TextSecure in order to include in our dataset, at least one mobile banking app and one secure chat app due to the sensitivity of the information they transmit and their claims of security. We choose Amazon Local in order to investigate whether there are implementation differences in apps from the same developers.[6]

The total number of downloads for the 100 selected apps amounts to over 10 billion

[6]Amazon and Amazon Local did implement SSL differently as of our first tests but both implement pinning as of our second tests.

#Downloads (Millions)	#Apps
> 1,000	4
500 - 1000	6
100 - 500	31
50 - 100	19
10 - 50	37
< 10	3

Table 4.1: Distribution of examined apps by number of downloads.

according to statistics provided by the Google Play store. Table 4.1 summarizes the distribution of apps by number of downloads (at the time of our experiments). Note that according to AndroidRank [2], there are about 1,000 apps with 10M+ downloads around the time of our experiments, thus, our 100-app set roughly corresponds to 10% of all apps with 10M+ downloads. The complete list of apps we analyze is reported in Appendix A.1.

4.3 Results

In this section, we present the result of our analysis and in Section 4.4, discuss the implication of the results.

4.3.1 Static Analysis

Although we evaluate TLSDroid using a dataset comprising 111,324 apps obtained from Play Store and finding that 24,053 (resp., 22,697) and 1,089 (resp., 15,060) of them, respectively, contain vulnerable and potentially vulnerable `TrustManager`s (resp., `HostnameVerifier`s), we do not discuss the results here as it is work in progress. Here, we focus on the results of our manual analysis only.

As discussed earlier, after decompiling an app, we search for SSL keywords to identify apps using SSL. We find that 93/100 apps include SSL code, while the remaining 7 (AVG antivirus, Candy Crush, Clean Master, Google Earth, Play Music, Jobsearch, and Voice Search) do not include SSL, even though they use tokens from Google, Facebook, or Twitter

TrustManager	SocketFactory
NaïveTrustManager	NaïveSslSocketFactory
BogusTrustManagerFactory	AndroidSSLSocketFactory
FakeX509TrustManager	FakeSocketFactory
TrustEveryoneTrustManager	TrustNonFacebookSocketFactory
TrustAllManager	TrustAllSSLSocketFactory
EasyX509TrustManager	EasySSLSocketFactory
IgnoreCertTrustManager	SimpleSSLSocketFactory
TrivialTrustManager	AllTrustingSSLSocketFactory
	BurstlySSLSocketFactory
	SelfSignedCertSocketFactory
	KakaoSSLSocketFactory
	TrustingSSLSocketFactory
	ConvivaSSLSocketFactory
	SSLSocketFactoryTrustAll

Table 4.2: Known vulnerable TrustManager and SocketFactory subclasses accepting all certificates [82].

user accounts on the device to access secure web pages. We find that 84% (78/93) of the apps with SSL code implement their own `TrustManager` or `HostnameVerifier`, while the remaining 16% use Android's default `TrustManager` and a combination of the subclasses of the `HostnameVerifier`.

We analyzed the 93 SSL-enabled apps and find that 48 of them include `HostnameVerifiers` accepting all hostnames. Out of the 48 apps, 41 implement hostname verifiers that always return true and/or use the `AllowAllHostnameVerifier` subclass, while the other 7 implement hostname verifiers that return true without any check. Our analysis also reveals that 46 of the SSL-enabled apps implement a `TrustManager` that accepts all certificates. Examples of `TrustManager` and `SocketFactory` doing so which have also been reported by prior work [82] are shown in Table 4.2.

4.3.2 Dynamic Analysis

We start our analysis by looking for information leakage, i.e., aiming to identify whether any sensitive information is sent in the clear, and then move on to MiTM attacks.

Sensitive Data	#Apps
Username and Password	3
GPS Location	4
IMEI Number	2
IMSI Number	1

Table 4.3: Sensitive data sent via HTTP.

Unencrypted Traffic. Table 4.3 summarizes the number of apps sending sensitive information unencrypted. Specifically, we find that 4shared, Duolingo, and Pic Collage send usernames and passwords in the clear during login, while Deezer encrypts the password with a nonce and transmits it over HTTP. Location is sent in the clear by MeetMe and Google Maps (no longer the case as of March 2015), GO SMS Pro/Launcher EX, and IMDB. Also, GO SMS Pro sends the IMEI and IMSI unencrypted, whereas Talking Angela sends the IMEI unencrypted. While it has been reported that transmission of IMEI/IMSI and location data are mostly done by embedded ad libraries [1, 73], this is only true for Talking Angela.

MiTM. Next, we use Fiddler to mount MiTM attacks in the three scenarios discussed in Section 4.2, starting with S1, i.e., an attacker having their CA certificate installed on the victim's device. We find that 91 apps establish login connections and grant access to secure pages, thus, allowing the attacker to capture sensitive information such as login credentials, financial information, contact list, calendar schedules, and chat messages. On the other hand, 9 apps do not connect thanks to SSL pinning. In scenario S2 (i.e., an attacker presenting an invalid certificate), 23 of the apps complete the connection (the implementation accepts all certificates), with 9 of them leaking sensitive information. In S3, when the attacker presents a certificate with a wrong CN and/or SubjectAltName, we find that 29 of the apps establish a connection, with 11 of them revealing sensitive information. A total of 20 apps are vulnerable in all three scenarios, with 9 of them revealing sensitive information.

During the MiTM attack, only 3 apps (and all in scenario S2) present the user with an

Analysis Type	Category	#Apps
Static Analysis	Analyzed Apps	100
	Apps with SSL code	93
	Accepts all certificates	46
	Accepts wrong hostname	48
Dynamic Analysis	Vulnerable to S1	91
	Vulnerable to S2	23
	Vulnerable to S3	29
	Vulnerable to S1, S2, and S3	20

Table 4.4: Summary of results.

error message, such as the one illustrated in Figure 4.1(a), indicating that the connection was refused due to an SSL certificate error. Other apps continue loading indefinitely, crash, display a message trying to redirect the user to a web browser, display a blank screen, or a generic message (e.g., "Unable to connect. Please check your connection and try again" and "Incorrect device time. Please ensure the device time is correctly set and try again"). Figure 4.1(b)–4.1(d) show examples of unhelpful warning messages displayed by some of the apps when the connection is not established.

4.4 Discussion

In this section, we analyze and discuss the implications of our results, recommendations, and the limitations of our study.

4.4.1 Analysis of Results

Summary of Results. The results of our analysis are summarized in Table 4.4. From the static analysis, we deduce that majority of SSL-enabled apps are vulnerable to MiTM attacks due to wrong hostname verification. Specifically, vulnerable hostname verifiers use the `AllowAllHostnameVerifier` class or return `true` without performing any checks (or checking if the common name ends with a given suffix). We also find that almost half of the

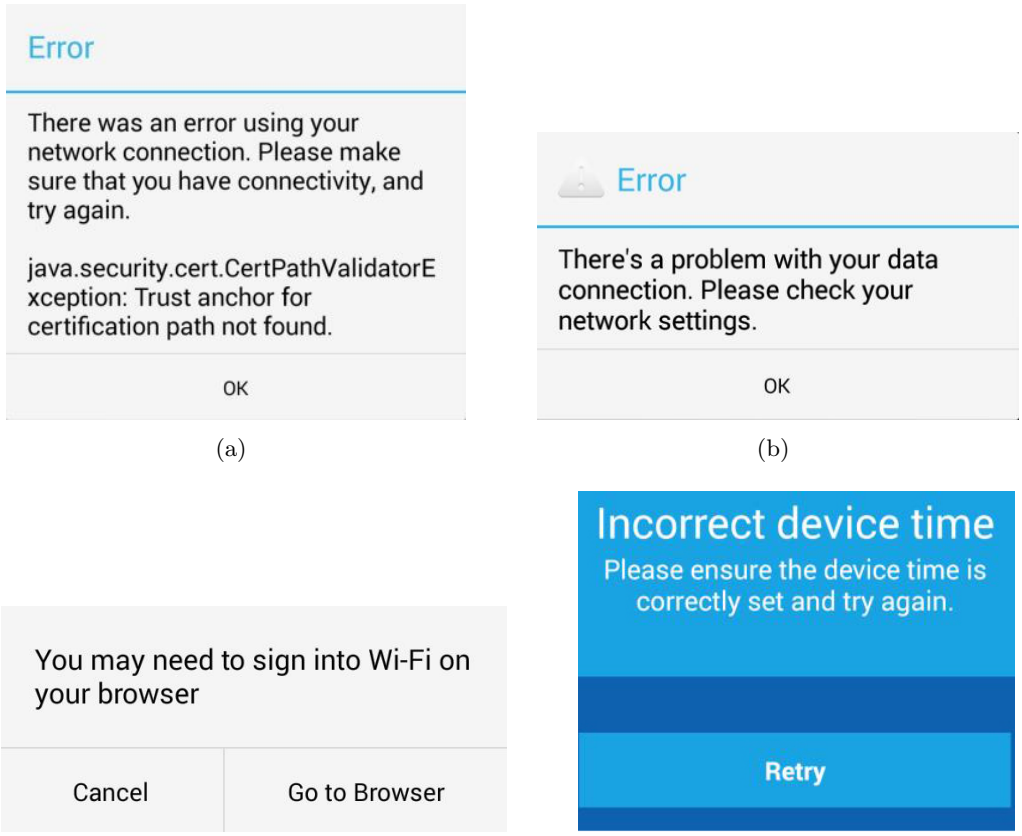


Figure 4.1: Warnings presented to users as a result of non-validation of an SSL certificate.

apps are vulnerable because their `TrustManager` accept invalid or self-signed certificates.

Then, following our dynamic analysis, we posit that in the presence of a MiTM attack: (1) apps with correct implementation of SSL pinning are not vulnerable; (2) apps with a vulnerable `TrustManager` are vulnerable; and (3) apps using `AllowAllHostnameVerifier` or a vulnerable `HostnameVerifier` are vulnerable. Also note that the overwhelming majority of apps (91) are vulnerable to powerful adversaries with a certificate on the user's device.

4.4.1.1 "Secure" Apps

We now analyze the 9 apps that do not establish connections in any of the three attack scenarios considered. These are: Amazon (10M downloads), Barclays Banking (1M), BBM (50M), Bitstrips (10M), Dropbox (100M), MeetMe (10M), TextSecure (500K), Twitter (100M), and Vine (10M), for a total of almost 300M downloads. Apps that are not vulnerable in all attack scenarios account for only 2.6% of the total number of downloads of the apps we test. While Twitter and Amazon apps are not vulnerable in any of our scenarios, they can be compromised indirectly using leaked credentials. This is because Tweetcaster (requires credentials for a Twitter account) and Amazon Local/Music (the same credentials as Amazon) are vulnerable in scenario S1.[7]

We also notice that 10 apps (e.g., Skype, Telegram, Viber) employ proprietary protocols or obscure their traffic, and/or did not connect via the MiTM proxy. These apps establish login sessions, and transmit messages/chat conversation in all three attack scenarios, except for Skype which does not load the contact list in scenario S3 (hence, no chat or call conversation can be initiated). Regardless, the in-app purchase SDK and the WebView interface (when present) of these apps (excluding Telegram, which does not offer in-app purchase) can be exploited in scenario S1 because e.g., debit card information used to subscribe and purchase call credit on Skype is retrievable.

4.4.1.2 Google Apps

Next, we investigate all the Google apps in our dataset that request full network access and have 10M+ downloads that are loaded on the Nexus 4 phone by default: Gmail, Calendar, Maps, Google+, Play Store, Play Music, and Play Movies. We find that they are all vulnerable in scenario S1, with debit card information and PayPal credentials exposed during in-app purchase. Note that this is not limited to the Google apps but also to non-Google apps that use Google's in-app purchase API (a total of 40 in our dataset). Other

[7] This is no longer the case for Amazon Local/Music as of March 2015.

sensitive information accessible from Google apps in scenario S1 include: usernames and encrypted passwords, email messages (from Gmail), location, calendar, and reminders.

4.4.1.3 Vulnerable Apps

With respect to apps vulnerable in any of the three attacking scenarios during dynamic analysis, we notice that 59 apps (including Netflix and Facebook[8]) are vulnerable in scenario S1 but not in S2 and S3. This means that these apps are secure against most adversaries, but not against very strong (e.g., state-like) adversaries. Protecting against vulnerability in S1 can be achieved using SSL pinning, which might however add some extra developmental cost. Also, there are 32 apps including, e.g., Groupon, Vevo, and Zoosk, that are vulnerable in scenario S2 and/or S3, i.e., they trust all certificates and/or all hostnames.

Five apps (Booking.com, IMDB, PayPal, Trip Advisor, and TuneIn Radio) establish login sessions in S3, however, we are not able to extract the credentials from the eavesdropped traffic. For instance, after login, PayPal refuses to establish further connections but displays a `Toast` with the message "SSL Error" at the bottom of the screen. We have two possible explanations for this. One is that logins are established simply due to cached cookies. Alternatively, we notice that developers tend to create several different `TrustManager` and `HostnameVerifier` implementing and enforcing different levels of security checks, thus potentially leading to oversights in creating connections with the wrong `TrustManager`.

A total of 20 apps are vulnerable in all three scenarios. One of such app is Job Search which displays a warning – shown in Figure 4.2 – signaling a problem with the certificate and providing users with the option to continue. As highlighted in prior work [193], users tend to ignore SSL warnings, and if users decide to continue, the adversary would access credentials and other sensitive data.

As mentioned earlier, a few apps send login credentials unencrypted. Specifically,

[8]Friend list, chat messages, and credentials are accessible from Facebook in S1.

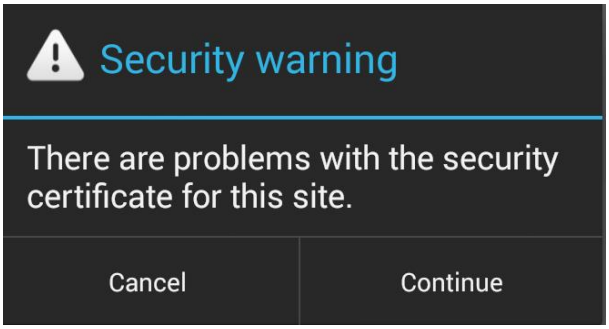


Figure 4.2: Certificate warning displayed by Job Search app in S2 and S3.

4shared sends login credentials as part of the URI in the GET command, thus making them trivially accessible by anyone eavesdropping on the traffic. Deezer encrypts user's password with a nonce before transmission, while Duolingo and Pic Collage send it in plaintext. While some developers may not consider their service to be sensitive, password reuse makes this practice extremely dangerous nonetheless, as discussed in prior works [67, 122].

Disclosure & Updates. We communicated our findings to the developers of apps transmitting sensitive information unencrypted and with possible vulnerabilities in scenarios S2 and S3. As a result, Pic Collage now establish login sessions and transmit user information via HTTPS, while 4shared hashes passwords before transmitting them over HTTP. Also, besides updates to Google Maps and Amazon apps discussed earlier, we also report that, as of April 2015, Ask.fm, Textplus and Viber no longer establish connection in scenario S2, while Groupon and IMDB do not in scenario S3.

4.4.1.4 Static vs Dynamic Analysis

As the results of our static and dynamic analysis differ somewhat, we set to investigate the reasons for this inconsistency.

Possible False Negatives in Dynamic Analysis. First, we note that dynamic analysis may miss some vulnerabilities discovered in static analysis i.e., vulnerable `TrustManagers` and `SocketFactory` implementations. This may happen as a result of the vulnerabilities being in apps' functionalities that are not tested during our dynamic analysis. For instance, static

analysis reveals that `NaïveTrustManager` and `FakeSocketFactory` interfaces are used in 7 apps by the Application Crash Reports for Android (ACRA) library. However, our dynamic analysis does not test ACRA interaction as no app crashed during usage, thus, it does not report these apps as vulnerable.

Possible False Positives in Static Analysis. A total of 23 apps possibly contain vulnerable `TrustManager` and `SocketFactory` implementations that dynamic analysis did not report. While 9 are confirmed vulnerabilities, the discrepancy suggests they are not used to create SSL sessions as they might have been introduced for testing purposes but never removed in production. Thus, this might generate some false positives during static analysis, however, because of code obfuscation, it is not always feasible to completely remove such false positives from static analysis.

4.4.2 Recommendations

We now discuss some recommendations to app developers vis-à-vis the vulnerabilities discussed in this chapter. We find that many `TrustManagers` are vulnerable as they accept self-signed certificates. This may occur as developers wish to accept self-signed certificates for testing purposes but forget to disable the feature in their production environment. While others purposely choose to employ self-signed certificates in production, and proceed to customize their `TrustManagers`. `TrustManagers` that accept self-signed certificates usually check for the presence of a single certificate, and verify whether or not it is valid by calling the *checkValidity()* method. This implies that any self-signed certificate currently valid would be accepted by the app.

In order to use self-signed certificates safely, developers should enable SSL pinning instead, rather than using a `TrustManager` that accepts all certificates. While the normal practice of using self-signed certificate is to verify the certificate thumbprint, we recommend the use of self-signed root certificates. Developers should create a keystore with self-signed root certificate to sign any number of leaf certificates to be employed on servers. The keystore

is then used to create a `TrustManager`. The leaf certificates are then verified against the self-signed root certificate. Credentials to be checked can include public key, SubjectAltName, signature, and any other field specified by the developer.

The same method should also be used when developers have a valid certificate signed by a trusted CA, but need to use a self-signed one for testing purposes. A separate keystore containing the self-signed root certificate for development environment should be created and used to initialize the `TrustManager`, but with the same pinning validation logic as that of the production environment. Employing the same pinning validation logic of checking public key, signature, etc., would ensure that developers only add a new self-signed root certificate to the development keystore and/or create new leaf certificates signed by the self-signed root certificate for any number of tests without disrupting the SSL validation logic. If one forgets to change the keystore used to initialize the `TrustManager`, the SSL pinning validation logic will act as a fail-safe default, leading to the failure of all secure connections, thus pointing the developer to check the keystore used for validation. A sample code snippet that could be used for this purpose is presented in Appendix A.2 and is also available online as a Github gist.[9] Similarly, to help developers detect vulnerable TLS implementations either from libraries they have included in their apps or those they have written themselves, we also make available our fork[10] of MalloDroid and will release TLSDroid soon.

As recent studies [15,91] show, the source of developers' information impact the security of their code. In a user study conducted by Acar et al. [15], developers who were only allowed access to Stack Overflow as their information source when they encountered coding challenges, produced significantly *less secure* but *more functional* code than those who were allowed to use only the official Android documentation. They show that of the 139 Stack Overflow threads their participants accessed, only 17% contain secure code. This result is confirmed by Fischer et al. [91], who show that of the 1.3 million apps from Play Store that

[9] `https://gist.github.com/luckenzo/416d9da141b302bfcfcd7b0d3b95489f`
[10] `https://github.com/luckenzo/mallodroid`

they analyze, 15.4% contain code snippet from Stack Overflow, with 97.9% of the snippet being insecure. As a result of these findings, it is important to provide developers with tools that help them detect vulnerabilities or with secure code snippet, which we do.

Finally, while the certificate pinning logic works when the peer certificate is signed by a known root CA, whenever a peer's certificate is not known beforehand, developers could use the default `TrustManager` that relies on the OS's trusted credentials.

4.4.3 Open Problems

Following our analysis, we highlight a few open research problems. First, we emphasize how, more than 2 years after prior work drew attention to SSL implementations on mobile platforms, many popular apps still accept all certificates and wrong hostnames, and are vulnerable to MiTM attacks. We argue for the need to give developers more effective tools that can help them detect and fix issues *before* the app is in production, and not only ways to detect vulnerabilities *after the fact*. To this end, we have discussed how developers could safely use self-signed certificates, and more strategies could be experimented with [29, 83].

The analysis of private information leakage and SSL vulnerabilities should be part of the vetting process performed by app markets, such as Google Play. These already scan apps for malware, inappropriate content, system interference, etc. [106]. Alternatively, research efforts should be encouraged to do so in lieu of app markets. Considering that tools are already available that allow crawling of app markets [206] as well as methodologies for large-scale analysis of SSL vulnerabilities [189], the community could design a portal keeping track over time of such vulnerabilities and reporting them to the public. Ideally, this could also be integrated with results from large-scale dynamic analysis.

Another set of open problems relates to designing meaningful mechanisms for visual feedback. Recall that in the browser context, the lock icon informs users that their connection is secure and that extensive research has analyzed (and proposed improvements to) SSL warnings [16, 86, 193]. On the contrary, little has been done in the context of smartphone

apps. This prompts a number of challenges as it is not clear how to provide meaningful feedback and how to proceed with respect to warnings. Arguably, we should not rely on the user to fix problems the community is not able to, as users often have no idea as to what warnings actually mean or what is the right course of action. A possible research avenue is to contextualize the warnings: if the user is connected via Wi-Fi, they could receive a different set of warnings, following a *contextual security* approach [146]. A user that connects to Wi-Fi and gets a descriptive warning (e.g., the certificate of www.twitter.com is signed by Mallory, Inc. but was expected to be signed by Verisign) is more likely to disconnect from the Wi-Fi.

4.4.4 Limitations

While it is inherently hard to perform large-scale dynamic analysis, we acknowledge the limited size of our 100-app corpus. However, this actually represents a reasonably sized sample of popular Android apps – as per statistics from AndroidRank [2], roughly 10% of all apps with at least 10M downloads. Also, we did not use cookies as an attack vector to exploit vulnerabilities. As mentioned by Sun and Beznosov [192], developers include OAuth tokens in cookies and these were often accessible, thus, they could be used to exploit further attack scenarios. Finally, the different program analysis approaches typically have pros and cons (see Chapter 2.3). For example, dynamic analysis produces some false negatives as we might not monitor all possible SSL sockets that connect to servers, while static analysis may have revealed vulnerable codes that are not reachable during runtime because they are developmental code only, potentially leading to a handful of false positives.

4.4.5 Concluding Remarks

In this chapter, we investigated the implementation of SSL in Android apps. Our results showed that a worrying number of popular apps do not implement the security protocol correctly, consequently making their apps vulnerable to MiTM attacks. In particular, 23

and 29 apps (out of 100), respectively implement the `TrustManager` and `HostnameVerifier` incorrectly, while four apps do not even employ encryption. Although during dynamic analysis we require the adversary to have access to or control the Wi-Fi access point the victim's device connects to, previous studies [90, 169] show that this is trivial to achieve with the deployment of fake or rogue access points. Moreover, users often connect to public Wi-Fi hotspots that are not secured by any form of encryption [57, 111], allowing anyone connected to the same network to eavesdrop and manipulate the network packets.

Due to the ease of setting up rogue access points (e.g., an unsecured Wi-Fi access points at public places with a spoof SSID), apps with vulnerable SSL implementation expose the users' private and sensitive information to eavesdroppers and active adversaries. Even when legitimate Wi-Fi access points are protected by encryption, prior work also shows that an adversary can crack the Wi-Fi security using techniques such as key re-installation attacks [203], brute force and rainbow tables [103], and statistical analysis [30, 35]. Hence, incorrect implementation of security and privacy protocols such as SSL by app developers removes a layer of protection from users. Thus, our work highlights the importance of correct implementation of the SSL protocol.

Chapter 5

Experimental Analysis of Popular Anonymous, Ephemeral, and End-to-End Encrypted Apps

In this chapter, we address **RQ2**: Do apps that claim to provide security and privacy properties that protect user information actually do?

The work presented in this chapter has been published in the 2016 NDSS UEOP [158]

5.1 Motivation

5.1.1 The Research Problem

As social networking takes to the mobile world, smartphone apps provide users with ever-changing ways to interact with each other with some of them being used by over a billion users. Following Edward Snowden's revelations, reports of government snooping, and increasingly detrimental hacks, privacy and anonymity technologies have been increasingly often in the news, with a growing number of users becoming aware – loosely speaking – of privacy and encryption notions [155]. Service providers have rolled out, or have announced they will, more privacy-enhancing tools, e.g., support for end-to-end encryption [143] and HTTPS by default [219]. At the same time, a number of social networking and communication apps have entered the market, promising to offer security and privacy properties such as anonymity, ephemerality i.e., making messages between communicating parties available only for a set amount of time, and/or end-to-end encryption (E2EE). While it is not uncommon to stumble upon claims like "military-grade encryption" or "NSA-proof" [8] in

the description of these apps, few studies thus far have actually analyzed the guarantees they provide.

Prior works have studied the adoption of these apps, investigating the motivation for their use [166,187]. Pielot and Oliver [166] study the motivations behind the use of Snapchat by teenagers finding that teens use Snapchat as they are excited by the ephemerality, see fewer risks, and non-commitment to persistent messengers. Similarly, Shein [187] points out that users do not use ephemeral messaging because they have something to hide, rather, because they do not want to add digital artifacts to their digital "detritus". In 2014, the Electronic Frontier Foundation (EFF) provided a Secure Messaging Scorecard that attempted to evaluate "secure" messaging apps based on a number of criteria.[1] But this scorecard has since been archived as they conclude that "it wasn't possible for us to clearly describe the security features of many popular messaging apps, in a consistent and complete way, while considering the varied situations and security concerns of our audience." [45] To this end, work presented in this chapter is motivated by the need for a systematic study of a careful selection of apps that claim to provide selected properties only. That is, in this chapter, we address the research question, RQ2, summarized here as:

- Do the security and privacy claims of "privacy-enhancing" Android apps match reality?

5.1.2 The Research Roadmap

To address the research question, we first focus on three security and privacy properties, namely, anonymity, ephemerality, and E2EE. Second, we compile a list of 18 apps that offer E2EE, anonymity and/or ephemerality, focusing on 8 popular ones (Confide, Frankly Chat, Secret, Snapchat, Telegram, Whisper, Wickr, and Yik Yak). We review their functionalities and perform an empirical evaluation, based on static and dynamic analysis, aimed to compare the claims of the selected apps against results of our analysis.

[1] `https://www.eff.org/node/82654`

The rest of this chapter is structured as follows: Section 5.2 presents the methodology used in our analysis as well as the apps we analyze, while Section 5.3 presents the results. Finally, Section 5.4 discusses the implication of our results and the limitations of the study.

5.2 Methodology

In this section, we present the definition of the selected security and privacy properties as it relates to the apps in our dataset, description of the selected apps, and the experimental setup used to analyze the presence or absence of these properties.

5.2.1 App Corpus

5.2.1.1 Apps Selection

We start by building a list of apps that are categorized as "anonymous" on Product Hunt [5]. We then look at their descriptions and at similar apps on the Google Play, and focus on those described as offering end-to-end encryption, anonymity and/or ephemerality, as defined below:

- ***Anonymity:*** is defined as the property that a subject is not identifiable within a set of subjects, known as the anonymity set [165], e.g., as provided by Tor [194] for anonymous communications. In the context of this paper, the term anonymity will be used to denote that users are anonymous with respect to other users of the app or with respect to the app provider.

- ***End-to-End Encryption (E2EE):*** Data exchanged between two communicating parties is encrypted in a way that only the sender and the intended recipient can decrypt it, so, e.g., eavesdroppers and service providers cannot read or modify messages.

App	Launched	#Downloads	Type	Content	Anonymity	Ephemerality	E2EE	Social Links
20 Day Stranger	2014	Unknown	Temporary OSN	Photos and location	Yes	No	No	No
Armortext	2012	50–100K	Chat (Enterprise)	Text and files	No	User-defined	Yes	Yes
BurnerApp	2012	100–500K	Temporary numbers	Call and SMS	N/A	N/A	No	Yes
Confide	2014	100–500K	Chat	Text, documents, photos	No	**After message is read**	**Yes**	Yes
CoverMe	2013	100–500K	Chat	Text, voice, photos, videos	No	User-defined	Yes	Yes
Disposable Number	Unknown	100–500K	Temporary numbers	Call and SMS	N/A	N/A	No	Yes
Frankly Chat	2013	500K–1M	Chat	Text, pictures, videos, voice	Optional for group chat	**10s**	No	Yes
Secret	2014	5–10M	Anonymous OSN, Chat	Text, photos,	**Yes**	No	No	Yes/No
Seecrypt SC3	2014	10–50K	Chat	Text, voice, files	No	No	Yes	Yes
Silent Circle	2012	100–200K	Encrypted Phone	Call, SMS, files	No	User-defined	Yes	Yes
Snapchat	2011	100–500M	Transient OSN	Photos, videos	No	**1–10s**	No	Yes
Telegram	2013	50–100M	Chat	Text, photos, audio, videos, files, location	No	**Optional**	**Optional**	Yes
TextSecure	2010	500K–1M	Chat	Text, files	No	No	Yes	Yes
TigerText	2010	500K–1M	Chat	Text, files	No	User-defined	Yes	Yes
Vidme	2013	50–100K	Video Sharing	Videos	Yes	No	No	No
Whisper	2012	1–5M	Anonymous OSN, Chat	Text, photos	**Yes**	No	No	No
Wickr	2012	100–500K	Chat	Text, files, photos, audio, videos	No	**User-defined**	**Yes**	Yes
Yik Yak	2013	1–5M	Local Bulletin	Text	**Yes**	No	No	No

Table 5.1: Our first selection of 18 smartphone apps providing at least one among ephemerality, anonymity, or end-to-end encryption. N/A denotes 'Not Applicable'. Apps in bold constitute the focus of our analysis.

- ***Ephemerality:*** In cryptography, it denotes the property that encryption keys change with every message or after a certain period. Instead, here ephemerality is used to indicate that messages are not available to recipients on the user interface after a period of time [33]. For instance, in apps like Snapchat, messages "disappear" from the app (but may still be stored on the server) a few seconds after they are read.

Initial List of Apps. Our first list contains 18 apps, listed in Table 5.1, where we also report their first release date, number of downloads as reported by Google Play Store, the kind(s) of content that can be shared via the apps (e.g., text, videos, files), and whether the apps create persistent social links. Note that our first selection does not include popular apps like WhatsApp, since it attempts, but does not guarantee[2] to provide E2EE for all users [96].

[2]At the time of this study, i.e., Summer of 2015, WhatsApp did not guarantee E2EE for all its user base but this is no longer the case.

5.2.1.2 Selection Criteria

Final List of Apps. From our corpus, we focus on apps with the most downloads that offer ephemerality, anonymity, E2EE, or, preferably, a combination of them. We exclude Silent Circle and TigerText as they require, respectively, paid subscription and a registered company email. We reduce our selection to 8 apps: Confide, Frankly Chat, Secret, Snapchat, Telegram, Whisper, Wickr, and Yik Yak (bold entries in Table 5.1). Next, we provide an overview of their advertised functionalities, complementing information in the Table 5.1. (Descriptions below are taken from either Google Play Store or the apps' respective websites.)

Confide: offers end-to-end encryption and ephemerality. It allows users to share text, photos, and documents from their device and it integrates with Dropbox and Google Drive. It provides read receipts and notification of screenshot capture attempts. Messages are not displayed on the app until the recipient *wands* over them with a finger, so that only a limited portion of the message is revealed at a time. After a portion of the message is read, it is grayed out Screenshots are also disabled on Android. Messages that have not been read are kept on the server for a maximum of 30 days.

Frankly Chat: is a chat app that allows users to send ephemeral messages (text, picture, video or audio), anonymous group chats, and un-send messages that the recipient has not opened. Messages disappear after 10 seconds but users can pin their chats disabling ephemerality. Both parties do not need to have the app installed to receive messages. A link is sent to a recipient without the app via email and when clicked, reveals the message. Messages are deleted from the server after 24 hours – whether they are read or not.

Secret: *(discontinued May 2015)* lets users post anonymously to other *nearby* users. Users can view *secrets* from other locations but can only comment on those from their nearby location. Users can chat privately with friends and engage in a group chat with the chat history disappearing after a period of inactivity.

Snapchat: is an app that allows users send text, photos and videos that are displayed for 1 to 10 seconds (as set by the user) before they "disappear", i.e., they are no longer available to their friends. If the recipient takes a screenshot, the sender is notified. Users can also view *Stories*, i.e., a collection of snaps around the same theme, and a so-called *Discover*, i.e., accessing snaps from different selected editorials.

Telegram: is a messaging app that lets users exchange text, photos, videos, and files. It also provides users with an option to engage in a "secret chat", which provides E2EE and optional ephemerality. Senders are notified if the recipient takes a screenshot. Account information, along with all messages, media, contacts stored at Telegram servers are deleted after 6 months of login inactivity.

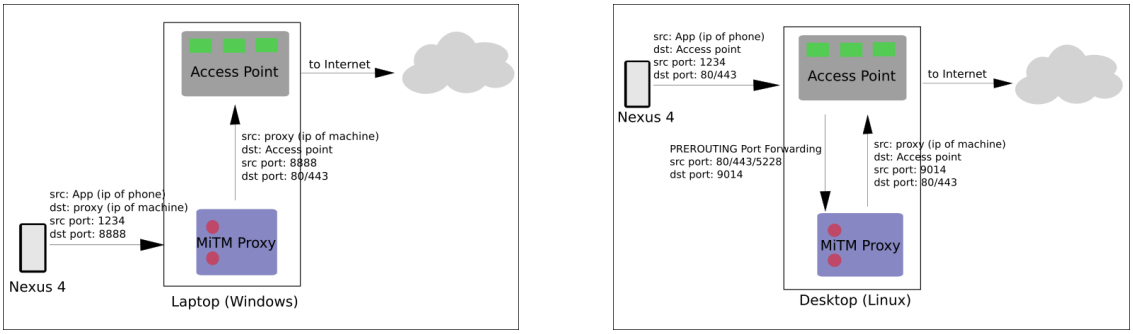
Whisper: is a location-based mobile social network that allows users to anonymously share texts displayed atop images, which are either selected by the users or suggested by the app. Users can view and respond to *whispers* either as a private message or via another *whisper*.

Wickr: is a chat app supporting text, audio, video, photos, and files, with user-defined ephemerality (maximum 6 days). It also allows users to engage in group chats, shred deleted files securely, and prevents screenshots on Android and claims to anonymize users by removing metadata (such as persistent identifiers or geo-location) from their contents.

Yik Yak: is a local bulletin-board social network allowing nearby users to post *yaks* anonymously. Users clustered within a 10-mile radius are considered local and can post, view, reply to, and up/down vote yaks but can only view *yaks* outside their locality.

5.2.2 Experimental Setup

Next, we present the setup we use to statically and dynamically examine the apps. We follow similar strategy presented in Chapter 4 and while the "reality" of an app's behavior is observed during dynamic analysis, statically examining apps may reveal potential vulnerabilities that may not be revealed due to limited code coverage during dynamic analysis.



(a) Proxy Setup Using Fiddler.

(b) Proxy Setup Using SSLsplit.

Figure 5.1: Dynamic analysis setup.

5.2.2.1 Static Analysis

Following the approach presented in Chapter 4.2, we use the same custom script, which allows us decompile .apk files to .jar files using dex2jar[3], from which we extract the related Java classes using JD-GUI.[4] We then search for SSL/TLS keywords like `TrustManager` [9], `HostnameVerifier` [3], `SSLSocketFactory` [6], and `HttpsURLConnection` [4]. Then, we manually inspect the `TrustManager` and `HostnameVerifier` interfaces used to accept or reject a server's credentials: the former manages the certificates of all Certificate Authorities (CAs) used in assessing a certificate's validity, while the latter performs hostname verification whenever a URL's hostname does not match the hostname in the certificate. We also emphasize that performing a manual analysis of the apps rather than using TLS-Droid introduced in Chapter 4.2, does not affect the results reported in Section 5.3.

5.2.2.2 Dynamic Analysis

Similar to the approach applied in Chapter 4.2.2, we manually instantiate the apps during dynamic analysis. Although, using a pseudorandom input generator such as Monkey [107] allows us automate the experiment and improves scalability, manually exercising apps allows us target specific features of the apps that employs the security and privacy properties

[3] `https://code.google.com/p/dex2jar/`
[4] `http://jd.benow.ca/`

we are interested in. We show in Chapter 7, how we can scale our analysis with real user input and automated pseudorandom input generator.

We conduct our experiments on a LG Nexus 4 smartphone running Android 5.1 and connected to a Wi-Fi access point under our control. (Note that the Wi-Fi network was secured using WPA2 to prevent unauthorized connections and ensure that only intended traffic was captured.) Our intention is to examine what an attacker can access from an app advertised as privacy-enhancing, and what can be deduced as regards privacy-enhancing claims. Hence, we assume an adversary that cannot elevate her privilege nor have access to a rooted device. We perform actions including sign-up, login, profile editing, sending/reading messages, while monitoring traffic transmitted and received by the apps. We collect traffic using Wireshark and analyze unencrypted traffic to check for sensitive information transmitted in the clear. We also rely on HTTP proxies such as Fiddler[5] and SSLsplit [7] to mount man-in-the-middle (MiTM) attacks and decrypt HTTPS traffic. Proxying is supported in two ways:

1. *Regular Proxy:* We install the Fiddler HTTP proxy on a Windows 8.1 laptop (which also acts as Wi-Fi access point), listening on port 8888, and manually configure the smartphone to connect to the proxy. Figure 5.1(a) illustrates our proxy setup using Fiddler. We also install Fiddler's CA certificate on the smartphone and laptop to allow for the decryption of HTTPS traffic.

2. *Transparent Proxy:* Some Android apps are programmed to ignore proxy settings, so Fiddler does not accept/forward their packets. This happens with Telegram, Wickr (non-CSS/JS), and Frankly Chat (chat packets). Therefore, we set up a transparent proxy as shown in Figure 5.1(b) using SSLsplit MiTM proxy set to listen on port 9014 on a Linux desktop running Fedora 22, which also acts as a Wi-Fi access point. We use *iptables* to redirect to port 9014 all traffic to ports 80, 443, and 5228 (GCM).

[5] `http://www.telerik.com/fiddler`

As SSLsplit uses a CA certificate to generate leaf certificates for the HTTPS servers each app connects to, we generate and install a CA certificate on the smartphone, and pass it to SSLsplit running on the Linux machine.

5.3 Results

Here, we report the results of our findings after statically and dynamically examining the apps and in the next chapter, discuss the implications of our results.

5.3.1 Static Analysis

Several sockets are usually created to transport data to different hostnames in an app, therefore, sockets in an app may have different SSL implementations. We observe different SSL implementations in the 8 apps, and summarize our findings below.

Non-Customized SSL Implementation. App developers can choose to use any one of five defined `HostnameVerifier` subclass for hostname verification, and use `TrustManager` initialized with a keystore of CA certificates trusted by the Android OS to determine whether a certificate is valid or not, or customize certificate validation by defining their own logic for accepting or rejecting a certificate. All the 8 apps in our corpus contain some non-customized SSL implementations. Telegram and Yik Yak only use non-customized SSL code, with the former relying on `BrowserCompatHostnameVerifier` class and building sockets from default `SSLSocketFactory`. Confide and Snapchat both use the `BrowserCompatHostnameVerifier` class, while Wickr has instances of all `HostnameVerifier` subclasses but uses the `BrowserCompatHostnameVerifier` class on most of its sockets. Snapchat does not customize its `TrustManager` either and registers a scheme from the default `SocketFactory`. Secret uses sockets from the default `SSLSocketFactory` but employs regex pattern matching for hostname verification.

Vulnerable TrustManager/HostnameVerifier. Frankly Chat, Whisper, and Wickr all con-

tain `TrustManager` and `HostnameVerifier` that accept all certificates or hostnames. Alas, this makes it possible for an adversary to perform MiTM attacks and retrieve information sent on the sockets that use the vulnerable `TrustManager` and/or `HostnameVerifier`. Vulnerable `HostnameVerifier` in Frankly Chat returns `true` without performing any check, while Wickr uses the `AllowAllHostnameVerifier` subclass which is also used in Whisper by Bugsense crash reporter.

Certificate Pinning. Confide, Frankly Chat, and Whisper implement certificate pinning. Confide pins the expected CA certificate which is also accessible from the decompiled apk, whereas Whisper uses the hash of the pinned certificate appended with the domain name to make certificate validation decisions. For Frankly Chat, a single certificate is expected and its hash is checked for in the received certificate chain. Frankly Chat also initializes another `TrustManager` with a keystore that loads certificate from file.

5.3.2 Dynamic Analysis

We now present the results of our dynamic analysis, which are also summarized in Table 5.2.

No Proxy. We start by simply analyzing traffic captured by Wireshark and observe that Secret and Frankly Chat send sensitive information in the clear. Specifically, in Frankly Chat, the Android advertising ID (a unique identifier) is transmitted in the clear, via an HTTP GET request, along with Device Name. The list of actions a user performs on Frankly Chat can also be observed from the request URL. Secret instead leaks Google Maps location requests (and responses) via HTTP GET.

Regular Proxy. Using Fiddler as a MiTM proxy, we notice that Confide and Whisper do not complete connection with their servers due to certificate pinning. Note that Whisper started implementing pinning after an update on April 22, 2015. Prior to that, one could capture Whisper traffic via Fiddler and access location and user ID. We also notice that Frankly Chat hashes passwords using MD5 without salt, while Snapchat sends usernames and passwords without hashing. Although inconsistently, Snapchat also sends previous

App	Fiddler	SSLsplit
Confide	No connection	No connection
Frankly Chat	TLS traffic is decrypted but packets containing chat messages not routed through proxy	TLS traffic is decrypted but there is no connection to the server when chat is attempted
Secret	All packets decrypted	Not Available (discontinued before we started using the transparent proxy)
Snapchat	All packets decrypted	All packets decrypted
Telegram	Connects but traffic does not pass through proxy	TLS traffic is decrypted but E2EE is enabled
Whisper	No connection	No connection
Wickr	Connects but traffic does not pass through proxy	TLS traffic is decrypted but E2EE is enabled
Yik Yak	All packets decrypted	All packets decrypted

Table 5.2: Summary of dynamic analysis results.

"expired" chat messages to the other party even though these are not displayed on the UI.[6]

Decrypted traffic from Secret, Whisper, and Yik Yak show that these apps associate unique user IDs to each user, respectively, *ClientId*, *wuid*, and *user ID*. We test the persistence of these IDs and find that, even if the apps' cache on the device is cleared through the Android interface, and the apps uninstalled and reinstalled, Whisper and Yik Yak retain the user ID from the uninstalled account and restore all previous *whispers* and *yaks* from the account. On Whisper, we manually delete its wuid and state files (in the /sdcard/whisper directory) before reinstalling the app: this successfully clears all previous *whispers* and a new wuid file is generated. However, it does not completely de-associate the device from the "old" account as the "new" account still gets notifications of private messages from conversations started by the "old" account. On the contrary, clearing Secret's cache unlinks previous messages, even without uninstalling the app.

Telegram and Wickr ignore the proxy settings, i.e., traffic does not pass through our proxy. Frankly Chat also ignore the proxy when sending chat messages but not for traffic generated by other actions.

Transparent Proxy. Using SSLsplit, we decrypt SSL-encrypted traffic from Wickr and Telegram. We do not find any sensitive information or chat messages being transmitted

[6]Note: we have informed Snapchat of this in October 2015 but did not get any response.

as the traffic is indeed encrypted. Apps for which SSL-encrypted traffic is recovered using Fiddler exhibit the same behavior on the transparent proxy, with Confide and Whisper not connecting due to pinning. We observe that certificate pinning is implemented on the socket used to transmit chat messages on Frankly Chat, as we cannot send chat messages but perform other actions, e.g., editing profiles and adding new friends. We also uninstall the CA certificate from the device to observe whether non-trusted certificate are accepted, and find that none of the apps established an HTTPS connection, which implies the apps do not use TrustMangers accepting any certificate as valid as reported in Chapter 4.

5.4 Discussion

We now discuss the implications of our analysis, in light of the properties promised by the 8 studied apps.

5.4.1 Anonymity

Anonymity w.r.t Other Users. Posts on Secret and Yik Yak (respectively, *secrets* and *yaks*) are not displayed along with any identifier, thus making users anonymous with respect to other users. On Whisper, due to the presence of a *display name* (non-unique identifier shown to other users) and its "Nearby" function, communities can be formed as a result of viewing and responding to *whispers* from nearby locations. Thus, it may be possible to link *whispers* to a display name, while at the same time querying the distance to the target, as described by Wang et al. [209].

Users who think they are anonymous are more likely to share sensitive content they might not share on non-anonymous OSN platforms, which makes "anonymous" apps potential targets of scammers/blackmailers that can identify users. This motivates us to examine the possibility of creating social links between users, i.e., linking a user and a set of actions. We find that this is not possible on Yik Yak as there are no one-to-one

conversations. Also, when the Yik Yak stream is monitored by a non-participating user, user IDs observed are symbolic to the real unique user ID. The symbolic user ID is only associated to one *yak*, hence, one cannot use it to link a user as the ID differs across *yaks* by the same user. Frankly Chat optionally offers k-anonymity during a group chat with $k+1$ friends. Due to the social link already present in the group (users chat with friends), psychological differences make it possible to identify who says what.

Anonymity w.r.t Service Provider. All apps associate identifiers to their users, which allows them to link each user across multiple sessions. Wickr claims to strip any metadata that could allow them to identify their users, thereby making users anonymous and impossible to track [211], but we cannot verify this claim since all traffic is end-to-end encrypted.

We observe different levels of persistence of user IDs in Secret, Whisper, and Yik Yak, as mentioned earlier. Secret stores identifiers on users' device, so an identifier would cease to persist beyond data and cache clearance. For Whisper and Yik Yak, we have two hypotheses as to why user IDs survive when the app is uninstalled and later reinstalled: either they store identifiers on their servers and restore them to a device on re-installation or they create the user IDs from the same device information using a deterministic function. This observation indicates that Whisper and Yik Yak's user IDs are linked to device information, thus making users persistently linkable. While Whisper and Yik Yak do reveal the information they collect from users in their privacy policy, previous work shows that the overwhelming majority of users do not read (or anyway understand) privacy policies [10]. Both apps collect information including device ID, IP address, geo-location, which can be used to track users. This, along with profiles from analytics providers (which both apps embed), can be used to de-anonymize users' age, gender, and other traits with a high degree of accuracy [185]. Finally, note that Whisper's description on Google Play, while including terms like 'anonymous profiles' and 'anonymous social network', is actually ambiguous as to whether they refer to anonymity with respect to Whisper or other users (or both).

Location Restriction. Secret and Yik Yak's restriction on feeds a user can see (and interact with) can simply be defeated, e.g., as Android lets users to use mock locations in developer mode. In combination with an app that feeds GPS locations chosen by the user (e.g., Fake GPS), this allows them to access geo-tagged messages from anywhere.

5.4.2 Ephemerality

Confide, Frankly Chat, Snapchat, Telegram, and Wickr offer message ephemerality with varying time intervals. Confide claims messages disappear after it is read once [60] but this is not the case as messages only "disappear" after a user navigates away. This implies the recipient can keep the message for longer as long as they do not navigate away from the opened message. In Frankly Chat, messages "disappear" after 10 seconds (even though users can pin messages). Ephemerality on Telegram only applies to "secret chats" and the expiration time is defined by the user. Snapchat and Wickr also let users determine how long their message last, with Snapchat defining a range of 1–10s (default 3s). On Snapchat, previous chat messages are actually part of the response received from the server, even though they are not displayed on the client's UI. This indicates that read messages are actually not deleted from Snapchat servers immediately, despite what is stated in Snapchat's privacy policy [188]. Since Confide and Frankly Chat implement certificate pinning, we cannot examine if responses from the server during chat contain past messages. Also, Telegram and Wickr encrypt data before transmission, thus we cannot make any analysis from intercepted packets.

Of all the apps offering ephemerality, only Confide and Wickr instruct the Android OS to prevent screen capture from a recipient. Obviously, however, the recipient can still take a photo with another camera, and video recording would defeat Confide's wand-based approach. Confide can claim to offer plausible deniability if a photo is taken, as messages are not displayed along with the name of the sender, hence, pictures would not preserve the link between the message and the identity of the sender. Frankly Chat, Snapchat, and Telegram

only notify the sender that the recipient has taken a screenshot, thus ephemerality claims are only valid assuming the recipient is not willing to violate a social contract between them and the sender. Also, if messages are not completely wiped from the server, the provider is obviously still subject to subpoena and/or vulnerable to hacking.

5.4.3 End-to-End Encryption

Confide and Wickr claim to employ E2EE by default, using AES-128 and AES-256, respectively. We can confirm E2EE in Wickr but not in Confide, since certificate pinning prevents interception of traffic. Also, Telegram offers E2EE for "secret chat" using AES-256 and client-server encryption (i.e. only the server and both clients can decrypt traffic) which also prevents MiTM attacks for non-secret chats. In both secret and non-secret chat, Telegram uses a proprietary protocol, MTProto, and transmit traffic over SSL although its webpage states otherwise.[7] Telegram and Wickr's implementations also claim to support perfect forward secrecy [198, 211].

5.4.4 Limitations

First, our analysis is only limited to eight apps which we plan to extend as part of future work. We downloaded the metadata of 1.4 million apps collected using PlayDrone [206] and found 455 apps that might be offering anonymity, ephemerality, or end-to-end encryption. Second, during dynamic analysis, we manually test apps, which consequently limits the number of apps we can analyze. However, manually testing the apps ensures that code paths that trigger network connectivity are tested compared to using automated tools (e.g., a pseudorandom input generator such as Monkey [107]) that may traverse many code paths that do not initiate any network connectivity. This limitation can be addressed by using Chimp [17] to crowdsource human input. We also plan to explore automation of the process, e.g., by using Monkey [107] or other input generators to test apps on the Android

[7] `https://core.telegram.org/mtproto#http-transport`

emulator.

5.4.5 Concluding Remarks

Although work presented in this chapter applies similar methodology as that in Chapter 4, it investigates a different problem. Specifically, some apps are advertised as "privacy-enhancing" or providing "military-grade encryption"; therefore, we investigate whether we can extract information or deanonymize users of such apps – actions the security or privacy properties these apps claim to provide should typically not allow us perform. Verifying whether apps advertised to exhibit one or more of these security and privacy properties is important due to the false sense of trust users put on them when they actually do not have these properties. For example, it is claimed that "Ephemeral messages are incredibly freeing and make people communicate more authentically and freely with their friends" [187]. While the "fun" and non-commitment of ephemeral messaging apps are drivers why users use them [166, 177].

When these apps fail to implement properties such as ephemerality correctly, users are exposed to privacy violations which may hinder the adoption of other apps that are advertised as secure and correctly implemented. In fact, our work show some interesting findings such as inconsistent application of ephemerality on ephemeral app, i.e., Snapchat and the use of persistent identifiers in *anonymous* apps. Finally, our findings (e.g., w.r.t E2EE) could also feed into any future scorecard (e.g., EFF Secure Messaging Scorecard) that attempts to provide security information guide to non-expert users.

Chapter 6

MaMaDroid: Detecting Android Malware by Building Behavioral Models using Abstracted API Calls

---

In this chapter, we address **RQ3**: Can we design new malware detection tools that are more robust to malware and Android framework evolution than the state-of-the-art?

The work presented in this chapter is the extended version of work published in NDSS [144] and it is currently under submission at ACM TOPS [160]

6.1 Motivation

6.1.1 The Research Problem

Malware running on mobile devices can be particularly lucrative, as it may enable attackers to defeat two-factor authentication for financial and banking systems [205] and/or trigger the leakage of sensitive information [108]. As a consequence, the number of malware samples has skyrocketed in recent years, and, due to its increased popularity, cybercriminals have increasingly targeted the Android ecosystem [59]. Detecting malware on mobile devices presents additional challenges compared to desktop/laptop computers: smartphones have limited battery life, making it impossible to use traditional approaches requiring constant scanning and complex computation [168]. Thus, Android malware detection is typically performed in a centralized fashion, i.e., by analyzing apps submitted to the Play Store using Bouncer [154] or Play Protect [65]. However, many malicious apps manage to avoid

detection [149, 213], and manufacturers as well as users can install apps that come from third party stores that may not perform any malware checks [227].

As a result, the research community has proposed a number of techniques to detect malware on Android. Previous work has often relied on the permissions requested by apps [77, 182], using models built from malware samples. This, however, is prone to false positives, since there are often legitimate reasons for benign apps to request permissions classified as dangerous [77]. Another approach, used by DroidApiMiner [14], is to perform classification based on API calls frequently used by malware. However, relying on the most common calls observed during training prompts the need for constant retraining, due to the evolution of malware and the Android API alike. For instance, "old" calls are often deprecated with new API releases, so malware developers may switch to different calls to perform similar actions. Hence, in this chapter we address RQ3 (see Section 1.1) which is divided into the following sub-questions:

Q1 Can we model the behavior of malicious apps in a way that improves malware detection?

Q2 Can we design a malware detection system that does not require frequent retraining of the classification model but that is still acceptably effective?

6.1.2 The Research Roadmap

To address Q1, we model the behavior of apps as Markov chains derived from the sequence of API calls in their executable files using static analysis. Our intuition behind the use of the sequences of API calls is that malware may use calls for different operations, and in a different order, than benign apps. For example, android.media.MediaRecorder can be used by any app that has permission to record audio, but the call sequence may reveal that malware only uses calls from this class *after* calls to getRunningTasks(), which allows recording conversations [220], as opposed to benign apps where calls from the class may

appear in *any* order. To measure the effectiveness of this approach, we compare it to a detection system that models apps using the frequency of API calls rather than sequences of the API calls (i.e., DroidApiMiner [14]).

We address Q2 by abstracting the API calls we use in our model to different levels of granularity. Abstraction provides resilience to API changes in the Android framework as for example, while method calls may change frequently (due to deprecation and addition) with new API releases, packages are added and removed less frequently. At the same time, the abstraction we apply does not "abstract away" the behavior of an app. For instance, packages include classes and interfaces used to perform similar operations on similar objects, so we can model the types of operations from the package name alone. For example, the `java.io` package is used for system I/O and access to the file system, even though there are different classes and interfaces provided by the package for such operations. We measure the contribution of abstraction to the effectiveness of our detection tool by evaluating the tool on dataset spanning several years, and comparing it to a system that employs similar modeling approach but without abstraction.

As outlined in Chapter 1.4, the work presented here is in collaboration with other co-authors, and they all contributed towards its completion. Specifically, the implementation of Markov chain-based modeling (Section 6.2.1.3), classification (Section 6.2.1.4), dataset characterization (Section 6.3.2), evaluation of MaMaDroid in 2 of 3 modes i.e., family and package modes (Section 6.4.1), and the runtime measurement of the modeling and classification modules of MaMaDroid are the contributions of the other coauthors.

The rest of this chapter is organized as follows: Section 6.2 presents the design and implementation of MaMaDroid and its variant FAM, while Section 6.3 introduces the dataset used to evaluate MaMaDroid and FAM. Section 6.4 presents the results achieved by MaMaDroid and FAM and finally, Section 6.5 discusses possible evasion techniques and the limitations of our work.

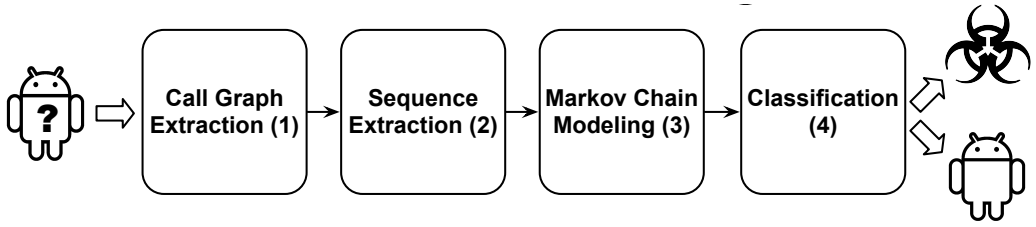


Figure 6.1: Overview of MaMaDroid's operation. In (1), it extracts the call graph from an Android app, next, it builds the sequences of (abstracted) API calls from the call graph (2). In (3), the sequences of calls are used to build a Markov chain and a feature vector for that app. Finally, classification is performed in (4), labeling the app as benign or malicious.

6.2 Design and Implementation of MaMaDroid

We now present our malware detection tool: MaMaDroid, its design and implementation. To measure the contribution of the Markov chain model derived from the sequence of API calls to the effectiveness of MaMaDroid, we build a variant (which we call FAM i.e., frequency analysis model) that uses the frequency of API calls rather than their sequences and compare both. To measure the contribution of abstraction to its effectiveness, we compare its variant FAM to DroidApiMiner since they are both based on the frequency of API calls. In this section, we introduce both MaMaDroid and its variant, FAM.

6.2.1 App Behavior as Markov Chains (MaMaDroid)

Building Blocks. MaMaDroid's operation goes through four phases, as depicted in Figure 6.1. First, we extract the call graph from each app by using static analysis (1), then, we obtain the sequences of API calls using all unique nodes after which we abstract each call to class, package, or family (2). Next, we model the behavior of each app by constructing Markov chains from the sequences of API calls for the app (3), with the transition probabilities used as the feature vector to classify the app as either benign or malware using a machine learning classifier (4). In the rest of this section, we discuss each of these steps in detail.

```
package com.fa.c;

import android.content.Context;
import android.os.Environment;
import android.util.Log;
import com.stericson.RootShell.execution.Command;
import com.stericson.RootShell.execution.Shell;
import com.stericson.RootTools.RootTools;
import java.io.File;

public class RootCommandExecutor {
  public static boolean Execute(Context paramContext) {
    paramContext = new Command(0, new String[] { "cat " + Environment.getExternalStorageDirectory().
         getAbsolutePath() + File.separator + Utilities.GetWatchDogName(paramContext) + " > /data/" +
         Utilities.GetWatchDogName(paramContext), "cat " + Environment.getExternalStorageDirectory().
         getAbsolutePath() + File.separator + Utilities.GetExecName(paramContext) + " > /data/" +
         Utilities.GetExecName(paramContext), "rm " + Environment.getExternalStorageDirectory().
         getAbsolutePath() + File.separator + Utilities.GetWatchDogName(paramContext), "rm " +
         Environment.getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities.
         GetExecName(paramContext), "chmod 777 /data/" + Utilities.GetWatchDogName(paramContext), "
         chmod 777 /data/" + Utilities.GetExecName(paramContext), "/data/" + Utilities.GetWatchDogName
         (paramContext) + " " + Utilities.GetDeviceInfoCommandLineArgs(paramContext) + " /data/" +
         Utilities.GetExecName(paramContext) + " " + Environment.getExternalStorageDirectory().
         getAbsolutePath() + File.separator + Utilities.GetExchangeFileName(paramContext) + " " +
         Environment.getExternalStorageDirectory().getAbsolutePath() + File.separator + " " +
         Utilities.GetPhoneNumber(paramContext) });
    try {
      RootTools.getShell(true).add(paramContext);
      return true;
    }
    catch (Exception paramContext) {
     Log.d("CPS", paramContext.getMessage());
    }
    return false;
  }
}
```

Figure 6.2: Code from a malicious app (com.g.o.speed.memboost) executing commands as root.

6.2.1.1 Call Graph Extraction

The first step in MaMaDroid is to extract the app's call graph. We do so by performing static analysis on the app's apk, i.e., the standard Android archive file format containing all files, including the Java bytecode, making up the app. We use a Java optimization and analysis framework, Soot [202], to extract call graphs and FlowDroid [26] to ensure contexts and flows are preserved. Specifically, we use FlowDroid, which is based on Soot, to create a dummy main method that serves as the main entry point into the app under analysis (AUA). We do so because Android apps have multiple entry points via which they can be started or invoked. Although apps have an activity launcher, which serves as the main entry

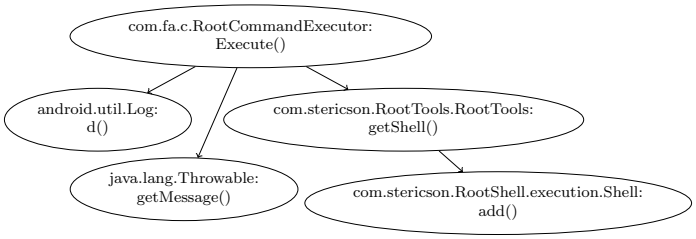


Figure 6.3: Call graph of the API calls in the try/catch block of Figure 6.2. (Return types and parameters are omitted to ease presentation).

point, it is not mandatory that they are implemented (e.g., apps that run as a service), hence, creating a single entry point allows us to reliably traverse the AUA. FlowDroid also lets us model the information flow from sources and sinks using those provided by SuSi [171] as well as callbacks.

To better clarify the different steps involved in our system, throughout this section, we employ a "running example" using a real-world malware sample. Figure 6.2 lists a class extracted from the decompiled apk of malware disguised as a memory booster app (with package name com.g.o.speed.memboost), which executes commands (rm, chmod, etc.) as root.[1] To ease presentation, we focus on the portion of the code executed in the try/catch block. The resulting call graph of the try/catch block is shown in Figure 6.3. For simplicity, we omit calls for object initialization, return types and parameters, as well as implicit calls in a method. Additional calls that are invoked when getShell(true) is called are not shown, except for the add() method that is directly called by the program code, as shown in Figure 6.2.

6.2.1.2 Sequence Extraction and Abstraction

In its second phase, MaMaDroid extracts the sequences of API calls from the call graph and abstract the calls to one of three mode.

Sequence Extraction. Since MaMaDroid uses static analysis, the graph obtained from Soot represents the sequence of functions that are potentially called by the app. However,

[1] `https://www.hackread.com/ghost-push-android-malware/`

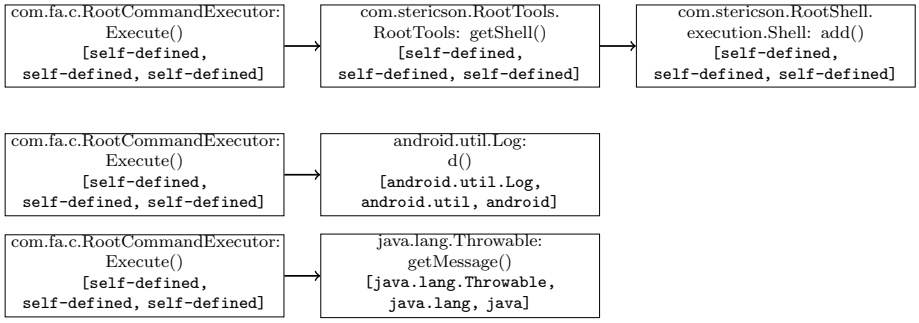


Figure 6.4: Sequence of API calls extracted from the call graphs in Figure 6.3, with the corresponding class/package/family abstraction in square brackets.

each execution of the app could take a specific branch of the graph and only execute a subset of the calls. For instance, when running the code in Figure 6.2 multiple times, the Execute method could be followed by different calls, e.g., getShell() in the try block only or getShell() and then getMessage() in the catch block.

Thus, in this phase, MaMaDroid operates as follows. First, it identifies a set of entry nodes in the call graph, i.e., nodes with no incoming edges (for example, the Execute method in the snippet from Figure 6.2 is the entry node if there is no incoming edge from any other call in the app). Then, it enumerates the paths reachable from each entry node. The sets of all paths identified during this phase constitutes the sequences of API calls which will be used to build a Markov chain behavioral model and to extract features. In Figure 6.4, we show the sequence of API calls obtained from the call graph in Figure 6.3. We also report in square brackets, the family, package, and class to which the call is abstracted.

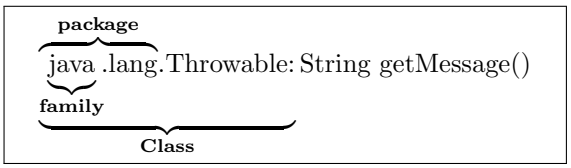


Figure 6.5: An example of an API call and its family, package, and class.

API Call Abstraction. Rather than analyzing raw API calls from the sequence of calls, we build MaMaDroid to work at a higher level, and operate in one of three modes by abstracting each call to its family, package, or class. Note that "family" as used in this

work refers to the "root" name of an API package and not to a "malware family" since we do not attempt to label each malware sample to its family. The intuition for this abstraction is to make MaMaDroid resilient to API changes and achieve scalability. In fact, our experiments presented in Section 6.3.2, show that from a dataset of 44K apps, we extract more than 10 million unique API calls, which, depending on the modeling approach used to model each app, may result in the feature vectors being very sparse. In Figure 6.5, we present an example of how abstraction works using the API call getMessage(), which is abstracted to, respectively, `java`, `java.lang`, and `java.lang.Throwable` in family, package, and class modes.

When operating in family mode, we abstract an API call to one of the nine Android families, i.e., `android`, `google`, `java`, `javax`, `xml`, `apache`, `junit`, `json`, `dom`, which correspond to the android.*, com.google.*, java.*, javax.*, org.xml.*, org.apache.*, junit.*, org.json, and org.w3c.dom.* packages. Whereas in package mode, we abstract the call to its package name using the list of Android packages from the documentation[2] consisting of 243 packages as of API level 24 (the latest version as of August 2016 when this work was done), as well as 95 from the Google API.[3] In class mode, we abstract each call to its class name using a whitelist of all class names in the Android and Google APIs, which consists respectively, 4,855 and 1116 classes.[4] In all modes, we abstract developer-defined (e.g., com.stericson.roottools) and obfuscated (e.g. com.fa.a.b.d) API calls respectively, as `self-defined` and `obfuscated`. Note that we label an API call as obfuscated if we cannot tell what its class implements, extends, or inherits, due to identifier mangling. Also, we first abstract an API call to its class name before abstracting to its family or package name. We do this to prevent the possibility of abstracting self-defined calls to those of the Google or Android framework when they are named in such a way that they look similar to that of the Android, java, or Google APIs. Overall, there are 11 (9+2) families, 340

[2] `https://developer.android.com/reference/packages.html`
[3] `https://developers.google.com/android/reference/packages`
[4] `https://developer.android.com/reference/classes.html`

(243+95+2) packages, and 5,973 (4,855+1,116+2) classes. Given the three different types of abstractions, naturally, we expect that the higher the abstraction, the lighter the system is, although possibly less accurate.

Reducing the Number of Classes. When operating in class mode, since there are 5,973 classes, processing the Markov chain transitions that results in this mode increases the memory requirements. Therefore, to reduce the complexity, we cluster classes based on their similarity. To this end, we build a co-occurrence matrix that counts the number of times a class is used with other classes in the same sequence in all datasets. More specifically, we build a co-occurrence matrix C, of size (5,973·5,973)/2, where $C_{i,j}$ denotes the number of times the i-th and the j-th class appear in the same sequence, for all apps in all datasets. From the co-occurrence matrix, we compute the cosine similarity (i.e., $cos(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}|| \cdot ||\boldsymbol{y}||}$), and use k-means to cluster the classes based on their similarity into 400 clusters and use each cluster as the label for all the classes it contains. Since we do not cluster classes abstracted to self-defined and obfuscated, we have a total of 402 labels.

6.2.1.3 Markov chain-Based Modeling

Next, MaMaDroid builds features vector used for classification, based on the Markov chains representing the sequences of abstracted API calls for an app. Before discussing this in detail, we first review the basic concepts of Markov chains.

Markov Chains. Markov chains are memoryless models where the probability of transitioning from a state to another only depends on the current state [153]. They are often represented as a set of nodes, each corresponding to a different state, and a set of edges connecting one node to another labeled with the probability of that transition. The sum of all probabilities associated to all edges from any node (including, if present, an edge going back to the node itself) is exactly 1. The set of possible states of the Markov chain is denoted as $\mathcal{S}$. If S_j and S_k are two connected states, P_{jk} denotes the probability of transition from S_j to S_k. P_{jk} is given by the number of occurrences (O_{jk}) of state S_k after

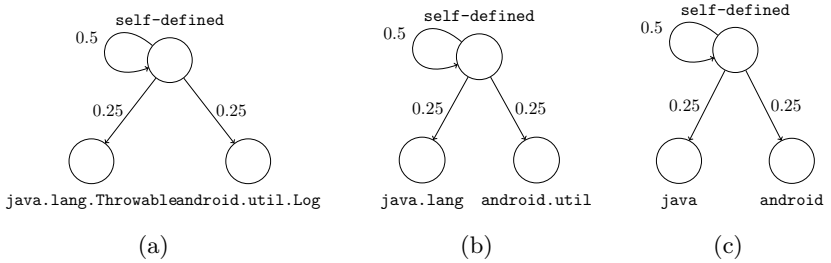


Figure 6.6: Markov chains originating from the call sequence in Figure 6.4 when using classes (a), packages (b) or families (c).

state S_j, divided by O_{ji} for all states i in the chain, i.e., $P_{jk} = \frac{O_{jk}}{\sum_{i \in \mathcal{S}} O_{ji}}$.

Building the Model. For each app, MaMaDroid takes as input the sequence of abstracted API calls of that app (classes, packages or families, depending on the selected mode of operation), and builds a Markov chain where each class/package/family is a state and the transitions represent the probability of moving from one state to another. For each Markov chain, state S_0 is the entry point from which other calls are made in a sequence. As an example, Figure 6.6 illustrates the Markov chains built using classes, packages, and families, respectively, from the sequences reported in Figure 6.4.

We argue that considering single transitions is more robust against attempts to evade detection by inserting useless API calls in order to deceive signature-based systems [129]. In fact, MaMaDroid considers all possible calls – i.e., all the branches originating from a node – in the Markov chain, so adding calls would not significantly change the probabilities of transitions between nodes (specifically, families, packages, or classes depending on the operational mode) for each app.

Feature Extraction. Next, we use the probabilities of transitioning from one state (abstracted call) to another in the Markov chain as the features vector of each app. States that are not present in a chain are represented as 0 in the features vector. The vector derived from the Markov chain depends on the operational mode of MaMaDroid. With families, there are 11 possible states, thus 121 possible transitions in each chain, while,

when abstracting to packages, there are 340 states and 115,600 possible transitions and with classes, there are 402 states therefore, 161,604 possible transitions.

Using PCA. We also apply Principal Component Analysis (PCA) [125], which performs feature reduction by transforming the feature space into a new space made of components that are a linear combination of the original features. The first components contain as much variance (i.e., amount of information) as possible. The variance is given as a percentage of the total amount of information of the original feature space. We apply PCA to the feature set in order to select the principal components. As PCA transforms the feature space into a smaller one where the variance is represented with as few components as possible, it considerably reduces computation/memory complexity. Also, PCA could improve the accuracy of the classification by removing from the feature space, features that make the classifier perform worse.

6.2.1.4 Classification

The last step is to perform classification, i.e., predicting apps as either benign or malware. To this end, we test MaMaDroid using different classification algorithms: Random Forests, 1-Nearest Neighbor (1-NN), 3-Nearest Neighbor (3-NN), and Support Vector Machines (SVM). Each model is trained using the features vector obtained from the apps in the training set and are evaluated using 10-fold cross validation. Also note that due to the different number of features used in different modes, we use two distinct configurations for the Random Forests algorithm. Specifically, when abstracting to families, we use 51 trees with maximum depth 8, while, with classes and packages, we use 101 trees of maximum depth 64. To tune Random Forests we follow the methodology presented by Bernard et al. [31].

6.2.2 App Behavior as Frequency of API Calls (FAM)

We also design a variant of MaMaDroid that is based on the frequency, rather than sequences, of abstracted API calls and in the rest of this chapter, we denote this variant as FAM (Frequency Analysis Model). We implement FAM to assess the role of abstraction in the detection performance of MaMaDroid. In the rest of this section, we discuss the steps that comprise FAM's operation.

6.2.2.1 API Call Extraction and Abstraction

API Call Extraction. The first step in FAM is to statically extract the API calls from an app. To do this, we use Androguard, a python tool for analyzing Android apps.[5] Using Androguard, we obtain the dalvik executable (DEX) of the app from which we extract the method and class names of every API call. As a result of comparing FAM to DroidApiMiner, we remove from the app, all API calls that are part of advertisement libraries using a manually compiled list of advertisement libraries obtained from the authors of DroidApiMiner.

After obtaining the API calls in an app, we abstract the API calls to their family, package, and class names following the same abstraction procedure in MaMaDroid already presented in Section 6.2.1.2.

6.2.2.2 Frequency Analysis-Based Modeling

In the second step of FAM, we model the behavior of each app by performing a frequency analysis and using abstracted calls that are more frequently used in malware than benign apps. That is, for each malware and benign dataset used for evaluation, we only use calls that are used more in the malware dataset than in the benign dataset. The number of calls used to model each app is dependent on the abstraction level/mode of operation i.e., family, package or class. In family mode, we use all families that occur more frequently in

[5] `https://github.com/androguard/androguard`

malware than in benign samples due to the small set of families (11). Whereas in package mode, we use the top 172 packages that occur more frequently in our malware dataset than the benign dataset. In class mode, we use the top 336 classes that occur more frequently in our malware dataset than in the benign. We use the top 172 packages and 336 classes so as to build the model with packages and classes, respectively, from at least two families (the android family has 171 packages) and packages (the `java.util` package has 335 classes).

6.2.2.3 Classification

The last step is the features vector extraction and classification of samples as either benign or malicious. We obtain as features vector, the usage/non-usage of the calls derived from step 2 (in previous section) in an app. We represent the presence/absence of e.g., a class in an app as 1/0. Finally, we use the Random Forests, 1-Nearest Neighbor (1-NN), and 3-Nearest Neighbor (3-NN) algorithms for classification and use the same Random Forests configuration presented in Section 6.2.1.4.

6.3 Dataset

In this section, we introduce the datasets used in the evaluation of MaMaDroid, FAM, and DroidApiMiner which include 43,940 apk files, specifically, 8,447 benign and 35,493 malware samples. We include a mix of older and newer apps, spanning from October 2010 to May 2016, as we aim to verify that MaMaDroid is robust to changes in Android malware samples as well as APIs. To the best of our knowledge, at the time of this study our dataset comprises the largest dataset of malware samples ever used in a research paper on Android malware detection.

Benign Samples. Our benign datasets consist of two sets of samples: (1) one, which we denote as `oldbenign`, includes 5,879 apps collected by PlayDrone [206] between April

Category	Name	Date Range	#Samples	#Samples (API Calls)	#Samples (Call Graph)
Benign	`oldbenign`	Apr 2013 – Nov 2013	5,879	5,837	5,572
	`newbenign`	Mar 2016 – Mar 2016	2,568	2,565	2,465
		Total Benign:	*8,447*	*8,402*	*8,037*
Malware	`drebin`	Oct 2010 – Aug 2012	5,560	5,546	5,512
	`2013`	Jan 2013 – Jun 2013	6,228	6,146	6,091
	`2014`	Jun 2013 – Mar 2014	15,417	14,866	13,804
	`2015`	Jan 2015 – Jun 2015	5,314	5,161	4,451
	`2016`	Jan 2016 – May 2016	2,974	2,802	2,555
		Total Malware:	*35,493*	*34,521*	*32,413*

Table 6.1: Overview of the datasets used in our experiments.

and November 2013, and published on the Internet Archive[6] on August 7, 2014; and (2) another denoted as `newbenign`, obtained by downloading the top 100 apps in each of the 29 categories on Play Store as of March 7, 2016, using the googleplay-api tool.[7] Due to errors encountered while downloading some apps, we only obtain 2,843 out of 2,900 apps. Note that 275 of these belong to more than one category, therefore, the `newbenign` dataset ultimately includes 2,568 unique apps.

Malware Samples. The set of malware samples includes apps that were used to test DREBIN [25], dating back to October 2010 – August 2012 (5,560), which we denote as `drebin`, as well as more recent ones that have been uploaded on the VirusShare[8] site over the years. Specifically, we gather from VirusShare, respectively, 6,228, 15,417, 5,314, and 2,974 samples from `2013`, `2014`, `2015`, and `2016`. We consider each of these datasets separately in our analysis.

6.3.1 Dataset Preprocessing

API Calls. For each app, we extract all API calls using Androguard since as explained in Section 6.4.4, these constitute the features used by DROIDAPIMINER [14] (against which

[6] `https://archive.org/details/playdrone-apk-e8`
[7] `https://github.com/egirault/googleplay-api`
[8] `https://virusshare.com/`

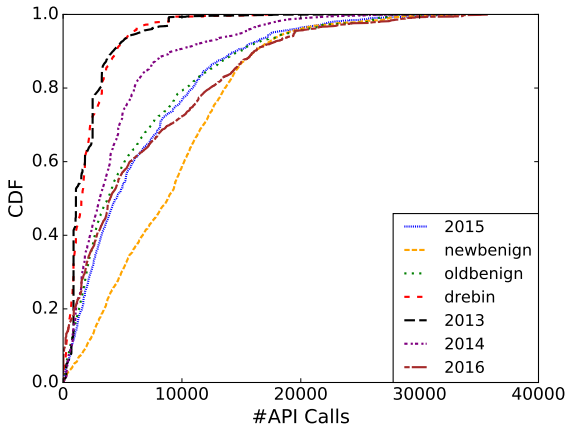
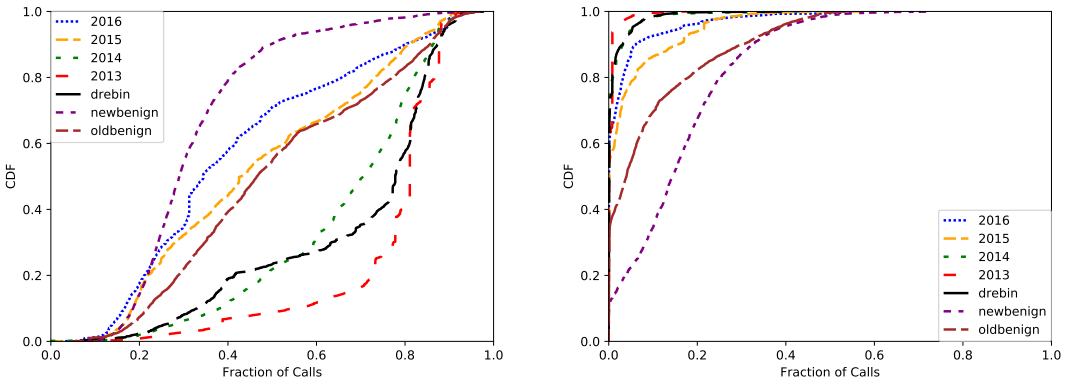


Figure 6.7: CDF of the number of API calls in different apps in each dataset.

we compare our system) as well as FAM. Due to Androguard failing to decompress some of the apks, bad CRC-32 redundancy checks, and errors during unpacking, we are not able to extract the API calls for all the samples, but only for 40,923 (8,402 benign, 34,521 malware) out of the 43,940 apps in our datasets.

Call Graphs. To extract the call graph of each apk, we use Soot. For some of the larger apks, Soot requires a non-negligible amount of memory to extract the call graph, so we allocate 16GB of RAM to the Java VM heap space. We find that for 2,472 (364 benign + 2,108 malware) samples, Soot is not able to complete the extraction due to it failing to apply the `jb` phase as well as reporting an error in opening some zip files (i.e., the apk). The `jb` phase is used by Soot to transform Java bytecode into jimple intermediate representation (the primary IR of Soot) for optimization purposes. Therefore, we exclude these apps in our evaluation and discuss this limitation further in Section 6.5.4.

In Table 6.1, we provide a summary of our seven datasets, reporting the total number of samples per dataset, as well as those for which we are able to extract the API calls (second-to-last column) and the call graphs (last column).



(a) `android` (b) `google`

Figure 6.8: CDFs of the percentage of `android` (a) and `google` (b) family calls in our dataset.

6.3.2 Dataset Characterization

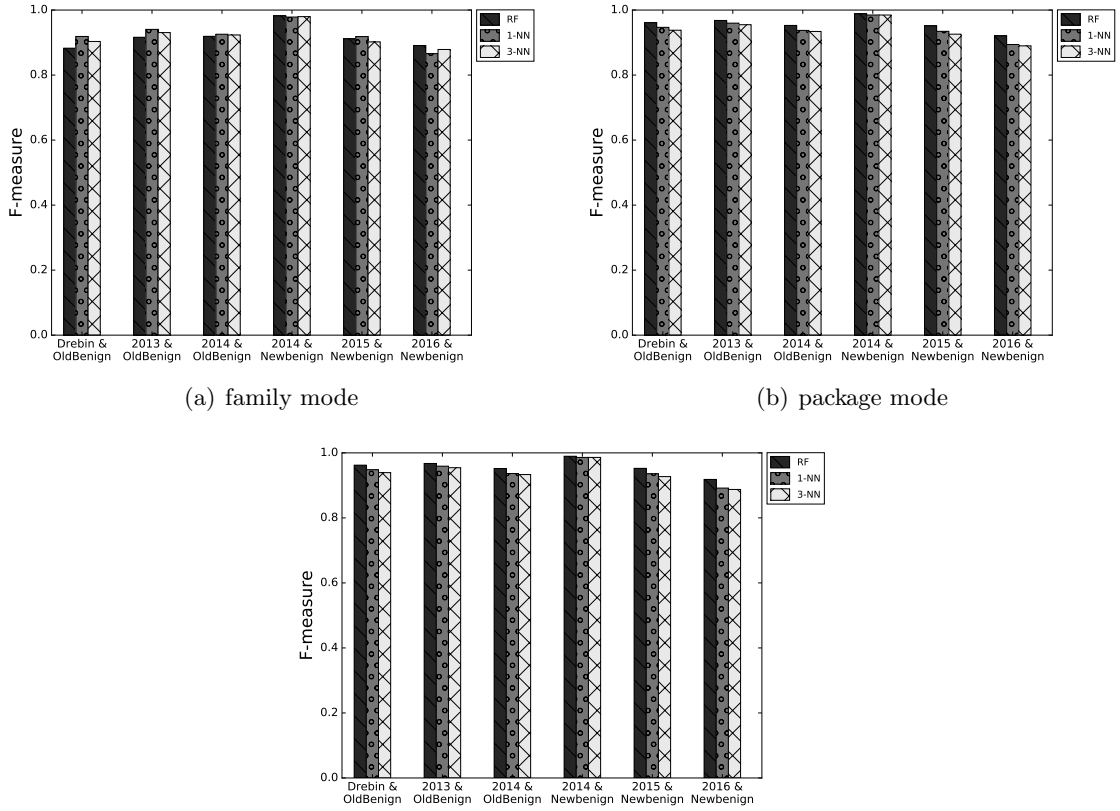
Aiming to shed light on the evolution of API calls in Android apps, we also performed some measurements over our datasets. In Figure 6.7, we plot the Cumulative Distribution Function (CDF) of the number of unique API calls in the apps in different datasets, highlighting that newer apps, both benign and malicious, use more API calls overall than older apps. This indicates that as time goes by, Android apps become more complex. When looking at the fraction of API calls belonging to specific families, we discover some interesting aspects of Android apps developed in different years. In particular, we notice that the usage of API calls in the `android` family become less prominent as time passes (Figure 6.8(a)), both in benign and malicious datasets, while `google` calls become more common in newer apps (Figure 6.8(b)). In general, we conclude that benign and malicious apps show the same evolutionary trends over the years. Malware, however, appears to reach the same characteristics (in terms of level of complexity and fraction of API calls from certain families) as legitimate apps with a few years of delay.

6.4 Evaluation

To assess the accuracy of the classification, we use the standard F-measure metric, calculated as $F - measure = 2 \cdot (\text{Precision} \cdot \text{Recall})/(\text{Precision} + \text{Recall})$, where Precision = TP/(TP+FP) and Recall = TP/(TP+FN). TP denotes the number of samples correctly classified as malicious, while FP and FN indicate, respectively, the number of samples mistakenly identified as malicious and benign. We perform 10-fold cross validation using at least one malicious and one benign dataset from Table 6.1. In other words, after merging the datasets, the resulting set is shuffled and divided into ten equal-size random subsets. Classification is then performed ten times using nine subsets for training and one for testing, and results are averaged out over the ten experiments.

6.4.1 Evaluating MaMaDroid

We now present an experimental evaluation of MaMaDroid in all (i.e., family, package, and class) modes. We use the datasets summarized in Table 6.1, and evaluate MaMaDroid, as per (1) its accuracy on benign and malicious samples developed around the same time; and (2) its robustness to the evolution of malware as well as of the Android framework by using older datasets for training and newer ones for testing and vice-versa. When implementing MaMaDroid in family mode, we exclude `json` and `dom` families because they are almost never used across all our datasets, and `junit`, which is primarily used for testing. In package mode, in order to avoid mislabeling when `self-defined` APIs have "android" in the name, we split the `android` package into its two classes, i.e., `android.R` and `android.Manifest`. Therefore, in family mode, there are 8 possible states, thus 64 features, whereas in package mode, 341 states and 116,281 features, and in class mode, 402 states and 161,604 features. In both modes, we do not report the result of SVM because it is less effective and efficient. For example, the F-Measure with SVM using Radial Basis Functions (RBF) is 0.09 lower than with Random Forests, and it is 5 times slower to



(a) family mode

(b) package mode

(c) class mode

Figure 6.9: F-measure of MaMaDroid classification with datasets from the same year using three different classifiers.

train in family mode than 3-Nearest Neighbors (the slowest among the other classification methods).

6.4.1.1 Detection Accuracy

We start by evaluating the performance of MaMaDroid when it is trained and tested on dataset from the same year. In Figure 6.9, we plot the F-measure achieved by MaMaDroid in family, package, and class modes using datasets from the same year for training and testing and the three different classifiers. As already discussed in Section 6.2.1.3, we apply PCA as it allows us transform a large feature space into a smaller one. For example when operating in package mode, PCA could be particularly beneficial to reduce computation

Mode / Dataset	[Precision, Recall, **F**-measure]																	
	Family			Family (PCA)			Package			Package (PCA)			Class			Class (PCA)		
drebin, oldbenign	0.82	0.95	0.88	0.84	0.92	0.88	0.95	0.97	**0.96**	0.94	0.95	0.94	0.95	0.97	**0.96**	0.92	0.94	0.93
2013, oldbenign	0.91	0.93	0.92	0.93	0.90	0.92	0.98	0.95	**0.97**	0.97	0.95	0.96	0.98	0.95	**0.97**	0.97	0.94	0.96
2014, oldbenign	0.88	0.96	0.92	0.87	0.94	0.90	0.93	0.97	**0.95**	0.92	0.96	0.94	0.93	0.97	**0.95**	0.91	0.95	0.93
2014, newbenign	0.97	0.99	0.98	0.96	0.99	0.97	0.98	1.00	**0.99**	0.97	1.00	**0.99**	0.98	1.00	**0.99**	0.98	1.00	**0.99**
2015, newbenign	0.89	0.93	0.91	0.87	0.93	0.90	0.93	0.98	**0.95**	0.91	0.97	0.94	0.93	0.98	**0.95**	0.92	0.97	0.94
2016, newbenign	0.87	0.91	0.89	0.86	0.88	0.87	0.92	0.92	**0.92**	0.88	0.89	0.89	0.91	0.92	**0.92**	0.88	0.92	0.90

Table 6.2: Precision, Recall, and F-measure obtained by MaMaDroid when trained and tested with dataset from the same year, with and without PCA.

and memory complexity, since MaMaDroid originally has to operate over 116,281 features that may result in sparse features vector and overfitting. Hence, in Table 6.2 we report the precision, recall, and F-measure achieved by MaMaDroid in family (small features space), package, and class (large features space) modes with and without the application of PCA using Random Forests classifier. We report the results for Random Forests only because it outperforms both 1-NN and 3-NN (Figure 6.9) while also being very fast.

The Effects of PCA. When operating in package mode, we find that only 67% of the variance is taken into account by the 10 most important PCA components, whereas in family mode, at least 91% of the variance is included by the 10 PCA Components. As shown in Table 6.2, the F-measure using PCA is only slightly lower (up to 3%) than when using the full feature set. Since the results with and without PCA are about the same, it suggests that MaMaDroid does not suffer from overfitting which may result from the high dimensionality of its feature space.

Family Mode. In Figure 6.9(a), we report the F-measure when operating in family mode for Random Forests, 1-NN and 3-NN. The F-measure is always at least 0.88 with Random Forests, and, when tested on the `2014` dataset, it reaches 0.98. With some datasets, MaMaDroid performs slightly better than with others. In general, lower F-measures are due to increased false positives since recall is always above 0.91, while precision might be lower due to the fact that malware datasets are larger than the benign sets.

Package Mode. When MaMaDroid runs in package mode, the classification performance improves, with F-measure ranging from 0.92 (`2016` and `newbenign`) to 0.99 (`2014` and

newbenign), using Random Forests. In Figure 6.9(b), we report the F-measure achieved in this mode using Random Forests, 1-NN, and 3-NN.

Class Mode. When operating in class mode, the classification performance improves compared to family mode and retains the same performance compared to package mode. In particular, it achieves F-measure ranging from 0.92 (2016 and newbenign) to 0.99 (2014 and newbenign), using Random Forests. In Figure 6.9(c), we report the F-measure achieved in this mode using the three (i.e., Random Forests, 1-NN, and 3-NN) different classifiers.

In general, MaMaDroid performs better in package and class modes in all datasets with F-measure ranging from 0.92 – 0.99 compared to 0.88 – 0.98 in family mode. This is as a result of the increased granularity which enables MaMaDroid identify more differences between benign and malicious apps. On the other hand, however, this likely reduces the efficiency of the system, as many of the states derived from the abstraction are used only a few times. The differences in time performance between the two modes are analyzed in detail in Section 6.4.5. Overall, we find that abstraction to class (finest granularity) does not provide significantly higher accuracy when compared to abstraction to package, yielding an average increase in F-measure of 0.0012.

6.4.1.2 Detection Over Time

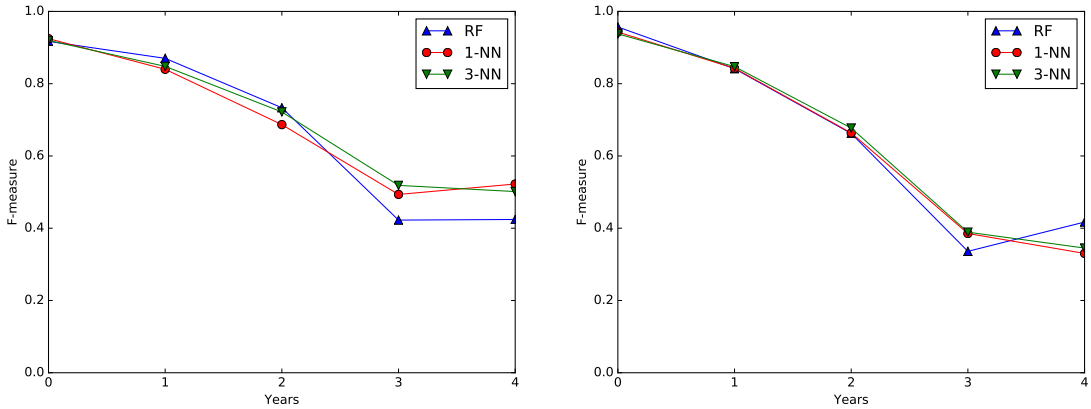
As Android evolves over the years, so do the characteristics of both benign and malicious apps. Such evolution must be taken into account when evaluating Android malware detection systems, since their accuracy might significantly be affected as newer APIs are released and/or as malicious developers modify their strategies in order to avoid detection. Evaluating this aspect constitutes one of our research questions, and one of the reasons why our datasets span across multiple years (i.e., 2010 – 2016).

Recall that MaMaDroid relies on the sequence of API calls extracted from the call graphs and abstracted to one of three modes of operation. Therefore, it may be less susceptible to changes in the Android API than other classification systems such as Droid-

ApiMiner [14] and Drebin [25]. Since these rely on the use, or the frequency, of certain API calls to classify malware vs benign samples, they need to be retrained following new API releases. On the contrary, retraining is not needed as often with MaMaDroid, since families and packages represent more abstract functionalities that change less over time. Consider, for instance, the `android.os.health` package: released with API level 24, it contains a set of classes that helps developers track and monitor system resources.[9] Classification systems built before this release – as in the case of DroidApiMiner [14] (released in 2013, when Android API was up to level 20) – need to be retrained if this package is more frequently used by malicious apps than benign apps, while MaMaDroid only needs to add a new state to its Markov chain when operating in package mode, while no additional state is required when operating in family mode.

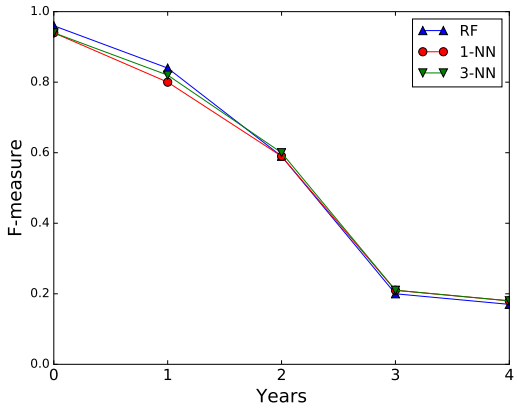
Older Training, Newer Testing. To verify this hypothesis, we evaluate MaMaDroid using older samples (i.e., `oldbenign` with each of the three oldest malware datasets – `drebin`, `2013`, and `2014`) as training sets and newer ones (i.e., `oldbenign` and all malware datasets newer than the one used in training) as testing sets. Figure 6.10(a) reports the average F-measure of the classification in this setting when MaMaDroid operates in family mode. The x-axis reports the difference in years between training and testing malware sets. We obtain F-measure of 0.86 when we classify samples that are one year older than the samples on which we train. Classification is still relatively accurate, at 0.75, even after two years. Then, from Figure 6.10(b), we observe that the average F-measure does not significantly change when operating in package mode. In class mode, we obtain average F-measure of 0.84 and 0.59, respectively (see Figure 6.10(c)), when trained on datasets one and two years older than the testing set. All modes of operation are affected by one particular condition, already discussed in Section 6.3.2: our benign datasets seem to be "ahead" of malicious ones by 1–2 years in the way they use certain API calls. As a result, we notice a drop in accuracy when classifying future samples and using `drebin` (with samples from 2010

[9] `https://developer.android.com/reference/android/os/health/package-summary.html`



(a) family mode

(b) package mode



(c) class mode

Figure 6.10: F-measure achieved by MaMaDroid using *older* samples for training and *newer* samples for testing. The x-axis shows the difference in years between the training and testing dataset.

to 2012) or `2013` as the malicious training set and `oldbenign` (late 2013/early 2014) as the benign training set. More specifically, we observe that MaMaDroid correctly detects benign apps, while it starts missing true positives and increasing false negatives – i.e., achieving lower recall.

Overall, we find that finer-grained abstraction actually performs worse with time when older samples are used for training and newer ones for testing. We note that this is due to a possible number of reasons: 1) newer classes or packages in recent API releases cannot be

captured in the behavioral model of older tools, whereas families are; and 2) evolution of malware either as a result of changes in the API or patching of vulnerabilities or presence of newer vulnerabilities that allow for stealthier malicious activities.

Newer Training, Older Testing. We also set to verify whether older malware samples can still be detected by the system – if not, this would obviously become vulnerable to older (and possibly popular) attacks. Therefore, we also perform the "opposite" experiment, i.e., training MaMaDroid with newer benign (March 2016) and malware (early 2014 to mid 2016) datasets, and checking whether it is able to detect malware developed years before. Specifically, Figure 6.11 reports results when training MaMaDroid with samples from a given year, and testing it with others that are up to 4 years older: it retains similar F-measure over the years. Specifically, in family mode (Figure 6.11(a)), it varies from 0.93 to 0.96, whereas in package mode (Figure 6.10(b)), from 0.95 to 0.97 with the oldest samples. In class mode, the average F-measure is 0.95 and 0.99, respectively, when trained with datasets one and two years newer than the test sets, as reported in Figure 6.11(c).

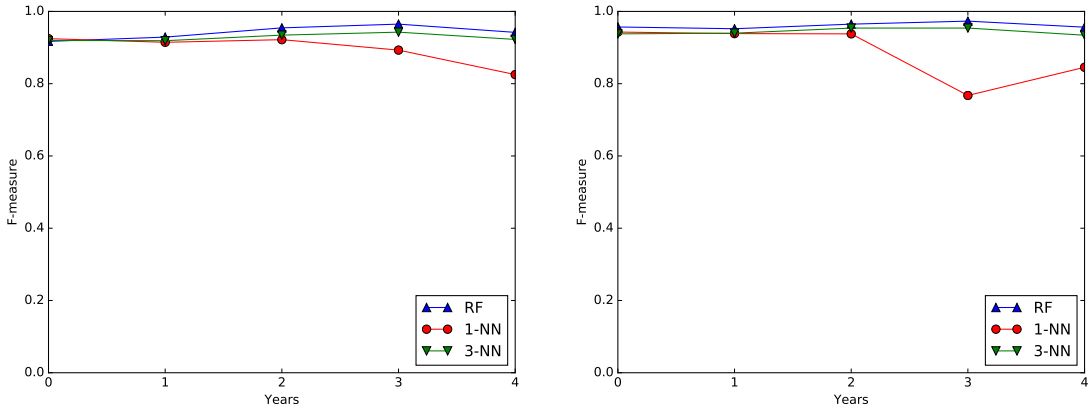
While MaMaDroid performs well in all modes, the finer the abstraction, the better it performs when trained with newer samples and tested on older samples.

6.4.2 Evaluating FAM

We again use the datasets in Table 6.1 to evaluate FAM's accuracy when training and testing on datasets from the same year and from different years.

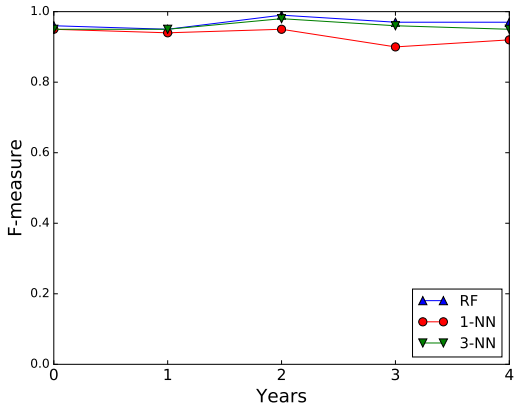
6.4.2.1 Detection Accuracy

We start our evaluation by measuring how well FAM detects malware by training and testing using samples that are developed around the same time. Figure 6.12 reports the F-measure achieved in family, package, and class modes using three different classifiers. Table 6.3 reports precision, recall, and F-measure achieved by FAM on each dataset combination, when operating in family, package, and class modes, using Random Forests. We



(a) family mode

(b) package mode



(c) class mode

Figure 6.11: F-measure achieved by MaMaDroid using *newer* samples for training and *older* samples for testing. The x-axis shows the difference in years between the training and testing dataset.

only report the results from the Random Forest classifier because it outperforms both the 1-NN and 3-NN classifiers.

Family Mode. Due to the number of possible families (i.e., 11), FAM builds a model from all families that occur more in our malware dataset than the benign dataset. Note that in this modeling approach, we also remove the `junit` family as it is mainly used for testing. When the `drebin` and `2013` malware datasets are used in combination with the `oldbenign` dataset, there are no families that are more frequently used in these datasets

Dataset \ Mode	[Precision, Recall, **F**-measure]								
	Family			Package			Class		
drebin, oldbenign	-	-	-	0.51	0.57	**0.54**	0.50	0.47	0.49
2013, oldbenign	-	-	-	0.53	0.57	0.55	0.58	0.57	**0.58**
2014, oldbenign	0.71	0.76	0.73	0.73	0.73	0.73	0.73	0.76	**0.75**
2014, newbenign	0.85	0.90	0.87	0.88	0.89	**0.89**	0.88	0.89	**0.89**
2015, newbenign	0.64	0.70	0.67	0.68	0.66	0.67	0.68	0.67	**0.68**
2016, newbenign	0.51	0.49	0.50	0.53	0.52	0.53	0.54	0.54	**0.54**

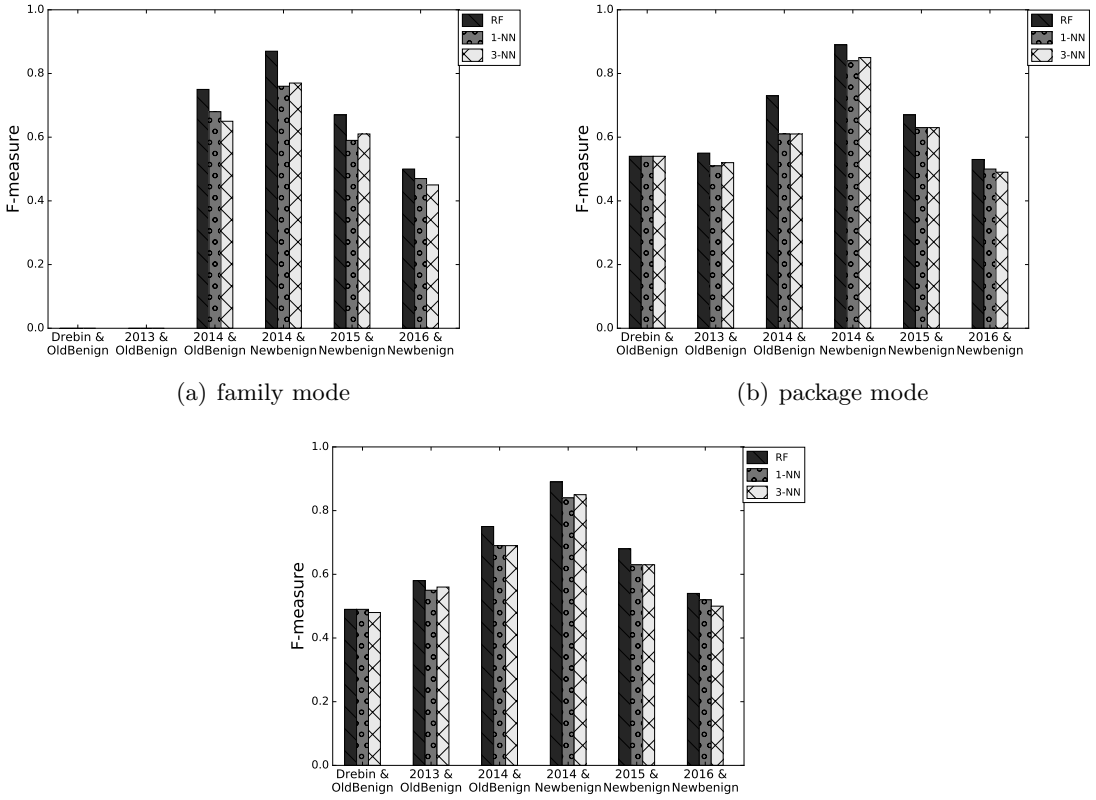
Table 6.3: Precision, Recall, and F-measure (with Random Forests) of FAM when trained and tested on dataset from the same year in all modes.

than the benign dataset. As a result, FAM does not yield any result with these datasets as it operates by building a model only from API calls that are more frequently used in malware than benign samples. With the other datasets, there are two (2016), four (2014), and five families (2015) that occur more frequently in the malware dataset than the benign one.

From Figure 6.12(a), we observe that F-measure is always at least 0.5 with Random Forests, and when tested on the 2014 (malware) dataset, it reaches 0.87. In general, lower F-measures are due to increased false positives. This follows a similar trend observed in Section 6.4.1.2.

Package Mode. When FAM operates in package mode, it builds a model using the minimum of, all packages that occur more frequently in malware or the top 172 packages used more frequently in malware than benign apps. We use the top 172 packages as we attempt to build the model where possible, with packages from at least two families (the android family has 171 packages). In our datasets, there are at least two (2013) and at most 39 (2016) packages that are used more frequently in malware than in benign samples. Hence, all packages that occur more in malware than benign apps are always used to build the model.

Classification performance improves in package mode, with F-measure ranging from 0.53 with 2016 and newbenign to 0.89 with 2014 and newbenign, using Random Forests. Figure 6.12(b) shows that Random Forests generally provides better results also in this



(a) family mode

(b) package mode

(c) class mode

Figure 6.12: F-measure achieved by FAM with datasets from the same year and the different modes of operation.

case. Similar to family mode, the `drebin` and `2013` datasets respectively, have only five and two packages that occur more than in the `oldbenign` dataset. Hence, the results when these datasets are evaluated is poor due to the limited amount of features.

Class Mode. In class mode, FAM builds a model using the minimum of, all classes that occur more frequently in malware than benign samples or the top 336 classes that occur more in malware than benign apps. In our datasets, there are at least three classes that occur more in malware (`2013`) than benign samples and at most, 862 classes that occur more frequently in malware (`2015`) than the benign dataset.

In Figure 6.12(c), we report the performance of FAM showing that it performs best

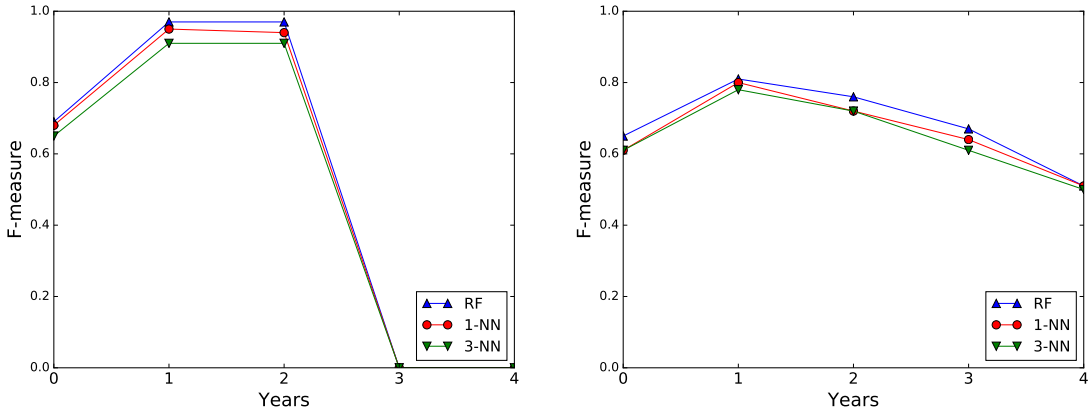
(F-measure of 0.89) with the `2014` and `newbenign` datasets.

6.4.2.2 Detection Over Time

Once again, we evaluate the detection accuracy over time, i.e., we train FAM using older samples and test it with newer samples and vice versa. We report the F-measure as the average of the F-measure over multiple dataset combinations; e.g., when training with newer samples and testing on older samples, the F-measure after three years is the average of the F-measure when training with, respectively, (`2015, newbenign`) and (`2016, newbenign`) and testing on `drebin` and `2013`.

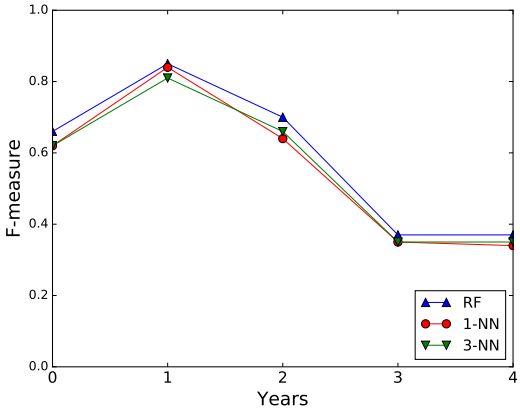
Older Training, Newer Testing. In Figure 6.13(a), we show the F-measure when FAM operates in family mode and is trained with samples that are older than the classified samples. The x-axis reports the difference in years between training and testing data. We obtain an F-measure of 0.97 when training with samples that are one year older than the samples in the testing set. As mentioned in Section 6.4.2.1, there are no results when the `drebin` and `2013` datasets are used for training, hence, after 3 years the F-measure is 0. In package mode, the F-measure is 0.81 after one year, 0.76 after two years, and it drops to 0.51 after four years (Figure 6.13(b)). Whereas in class mode, FAM achieves F-measure of 0.85 after one year and drops to 0.70 after two.

Although FAM appears to perform better in family mode than in package or class modes, note that the detection accuracy after one and two years in family mode does not include results when the training set is (`drebin, oldbenign`) or (`2013, oldbenign`) (see Section 6.4.2.1). We believe this is as a result of FAM performing best when trained on the `2014` dataset in all modes and performing poorly when trained with (`drebin, oldbenign`) and (`2013, oldbenign`) in package/class mode due to limited features. For example, result after two years (in family mode) is the average of the F-measure when training with (`2014, oldbenign/newbenign`) datasets and testing on the `2016` dataset. Whereas in package mode, the result is the average F-measure obtained from training with



(a) family mode

(b) package mode

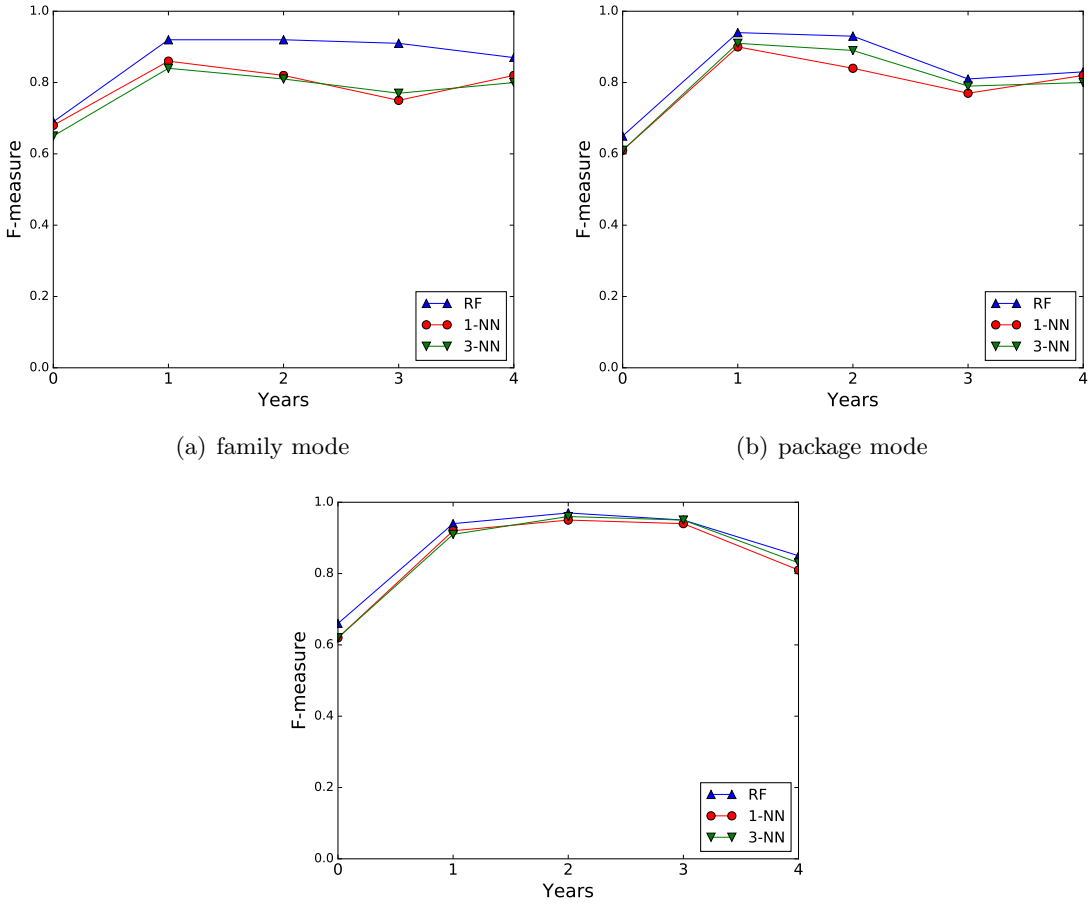


(c) class mode

Figure 6.13: F-measure achieved by FAM using *older* samples for training and *newer* samples for testing. The x-axis shows the difference in years between the training and test data.

(`drebin, oldbenign`), (`2013, oldbenign`), and (`2014, oldbenign/newbenign`) datasets and testing with respectively, `2014`, `2015`, and `2016`.

Newer Training, Older Testing. We also evaluate the opposite setting, i.e., training FAM with newer datasets, and checking whether it is able to detect malware developed years before. Specifically, Figure 6.14 report results when training FAM with samples from a given year, and testing it with others that are up to 4 years older showing that F-measure range from 0.69 to 0.92 in family mode, 0.65 to 0.94 in package mode, and 0.66 to 0.97 in class mode. Recall that in family mode, FAM is unable to build a model when `drebin` and



(a) family mode

(b) package mode

(c) class mode

Figure 6.14: F-measure achieved by FAM using *newer* samples for training and *older* samples for testing. The x-axis shows the difference in years between the training and test data.

2013 are used for training, thus, effecting the overall result. This effect is minimal in this setting since the training sets are newer than the testing sets. That is, the `drebin` dataset is not used to evaluate any dataset, while the 2013 dataset is used in only one setting i.e., when the training set is one year newer than the testing set.

6.4.3 The Effectiveness of the Modeling Approach: FAM vs MaMaDroid

Recall that with Q1, we attempt to improve the effectiveness of malware detection, as a result, we model the behavior of apps as Markov chains using the sequence of abstracted

Dataset \ Mode	F-measure			
	Family		Package	
	FAM	MaMaDroid	FAM	MaMaDroid
drebin, oldbenign	-	**0.88**	0.54	**0.96**
2013, oldbenign	-	**0.92**	0.55	**0.97**
2014, oldbenign	0.73	**0.92**	0.73	**0.95**
2014, newbenign	0.87	**0.98**	0.89	**0.99**
2015, newbenign	0.67	**0.91**	0.67	**0.95**
2016, newbenign	0.50	**0.89**	0.53	**0.92**

Table 6.4: F-measure of FAM and MaMaDroid in family and package modes when trained and tested on dataset from the same year.

API calls (i.e., MaMaDroid). In this section therefore, we compare the detection accuracy of MaMaDroid to FAM– a variant of MaMaDroid that is based on a frequency analysis model – to measure its effectiveness, focusing on the family and package modes.

Detection Accuracy of Malware from Same Year. In Table 6.4, we report accuracy of FAM and MaMaDroid when they are trained and tested on samples from the same year using Random Forests in both family and package modes. MaMaDroid outperforms FAM in all test settings with FAM achieving its best result when trained and tested with (2014 and newbenign), achieving F-measures of 0.89 (package mode). Overall, MaMaDroid achieves higher F-measure compared to FAM due to its modeling approach i.e., Markov chain modeling of the *sequence* of calls, as it can model differences between benign and malicious apps when popular API calls common to both invoke other API calls in different order. In addition, MaMaDroid is more robust as with some datasets, FAM fails to build a model when the abstracted calls occur equally or more frequently in benign samples compared to malicious samples.

Detection Accuracy of Malware from Different Years. We also compare FAM with MaMaDroid when they are trained and tested with datasets across several years. In Table 6.5, we report the F-measures achieved by MaMaDroid and FAM in package mode using Random Forests, and show how they compare using two different sets of experiments. In the first set of experiments, we train MaMaDroid and FAM using samples comprising

	Testing Sets									
	drebin, oldbenign		2013, oldbenign		2014, oldbenign		2015, oldbenign		2016, oldbenign	
Training Sets	FAM	MaMa	FAM	MaMa	FAM	MaMa	FAM	MaMa	FAM	MaMa
drebin,oldbenign	0.54	**0.96**	0.50	**0.96**	0.50	**0.79**	**0.50**	0.42	**0.51**	0.43
2013,oldbenign	0.90	**0.93**	0.55	**0.97**	**0.95**	0.74	**0.87**	0.36	**0.82**	0.29
2014,oldbenign	**0.95**	0.92	**0.99**	0.93	0.73	**0.95**	**0.81**	0.79	**0.82**	0.78
	drebin, newbenign		2013, newbenign		2014, newbenign		2015, newbenign		2016, newbenign	
Training Sets	FAM	MaMa	FAM	MaMa	FAM	MaMa	FAM	MaMa	FAM	MaMa
2014,newbenign	**0.99**	**0.99**	**0.99**	**0.99**	0.89	**0.99**	0.86	**0.89**	0.82	**0.83**
2015,newbenign	0.92	**0.98**	0.84	**0.98**	0.95	**0.99**	0.67	**0.95**	**0.91**	0.90
2016,newbenign	0.83	**0.97**	0.69	**0.97**	0.91	**0.99**	0.86	**0.93**	0.53	**0.92**

Table 6.5: F-Measure of MaMaDroid (MaMa) vs its variant using frequency analysis (FAM).

the `oldbenign` and one of the three oldest malware datasets (`drebin`, `2013`, `2014`) each, and test them on all malware datasets. FAM outperforms MaMaDroid in nine out of the 15 experiments, largely, when the training set comprises the `drebin`/`2013` and `oldbenign` datasets. Recall that when `drebin` and `2013` malware datasets are used for training FAM in package mode, only five and two packages (and at most, 39 packages in `2016`) respectively, are used to build the model. It is possible that these packages are the principal components that distinguishes malware from benign samples. In the second set of experiments, we train MaMaDroid and FAM using samples comprising the `newbenign` and one of the three recent malware datasets (`2014`, `2015`, `2016`) each, and test them on all malware datasets. In this setting, MaMaDroid outperforms FAM in all but one experiment where FAM is only slightly better.

Overall, we find that the Markov chain-based model achieves higher detection accuracy in both family and package modes when MaMaDroid is trained and tested on dataset from the same year (Table 6.4) and across several years (Table 6.5).

6.4.4 The Effectiveness of Abstraction: FAM vs DroidApiMiner

In order to address Q2, we propose abstracting the API calls and evaluate how our system performs using datasets spanning several years. To measure whether abstracting API calls contributes to the effectiveness of MaMaDroid and FAM, we compare the detection performance of FAM to prior work that uses API features for Android malware classification.

We compare specifically to DroidApiMiner [14] because: (i) it uses API calls and their parameters, and it also uses the same modeling approach as FAM to perform classification; (ii) it reports high true positive rate (up to 97.8%) on almost 4K malware samples obtained from McAfee and Genome [226], and 16K benign samples; and (iii) its source code has been made available to us by the authors. For reference on how both compare to MaMaDroid, we include the results of MaMaDroid in the tables but do not discuss them here, since we already compared MaMaDroid to FAM in Section 6.4.3.

In DroidApiMiner, permissions that are requested more frequently by malware samples than by benign apps are used to perform a baseline classification. Then, the system also applies frequency analysis on the list of API calls after removing API calls from ad libraries, using the 169 most frequent API calls in the malware samples (occurring at least 6% more in malware than benign samples). Finally, data flow analysis is applied on the API calls that are frequent in both benign and malicious samples, but do not occur by at least 6% more in the malware set. Using the top 60 parameters, the 169 most frequent calls change, and authors report a precision of 97.8%.

After obtaining DroidApiMiner's source code from the authors, as well as a list of packages (i.e., ad libraries) used for feature refinement, we re-implement the system by modifying the code in order to reflect recent changes in Androguard (used by DroidApiMiner for API call extraction). We extract the API calls for all apps in the datasets listed in Table 6.1, and perform a frequency analysis on the calls. Recall that Androguard fails to extract calls for about 2% (1,017) of apps, thus DroidApiMiner is evaluated over the samples in the second-to-last column of Table 6.1. We also implement classification, which is missing from the code provided by the authors, using k-NN (with k=3) since it achieves the best results according to the paper. We use 2/3 of the dataset for training and 1/3 for testing as implemented by the authors.

Detection Accuracy of Malware from Same Year. In Table 6.6, we report the results of DroidApiMiner and FAM (family and package modes) when they are trained and

Dataset \ Mode	F-measure				
	Family		Package		
	FAM	MaMaDroid	FAM	MaMaDroid	DroidApiMiner
drebin, oldbenign	-	**0.88**	0.54	**0.96**	0.32
2013, oldbenign	-	**0.92**	0.55	**0.97**	0.36
2014, oldbenign	0.73	**0.92**	0.73	**0.95**	0.62
2014, newbenign	0.87	**0.98**	0.89	**0.99**	0.92
2015, newbenign	0.67	**0.91**	0.67	**0.95**	0.77
2016, newbenign	0.50	**0.89**	0.53	**0.92**	0.36

Table 6.6: F-measure of FAM and MaMaDroid in family and package modes as well as, DroidApiMiner [14] when trained and tested on dataset from the same year.

tested on datasets from the same year. When FAM operates in family mode, it performs better than DroidApiMiner in two of the four datasets combination. In package mode, it outperforms DroidApiMiner in four out of six datasets combinations. DroidApiMiner only performs better than FAM when they are both trained and tested using the (2014 and newbenign), and (2015 and newbenign) datasets in both modes.

Detection Accuracy of Malware from Different Years. In Table 6.7, we report the results of DroidApiMiner compared to FAM on different combination of datasets in package mode. First, we train it using older dataset composed of oldbenign combined with one of the three oldest malware datasets each (drebin, 2013, and 2014), and test on all malware datasets. With this configuration, DroidApiMiner achieves its best result (F-measure of 0.62) when trained and tested on 2014 and oldbenign. The F-measure drops to 0.33 and 0.39, respectively, when tested on samples one year into the future and past. With the same datasets, FAM achieves an F-measure of 0.73 which increases to 0.81 and 0.99, respectively, one year into the future and past. In this set of experiments, FAM outperforms DroidApiMiner in all 15 experiments showing that it is more effective in detecting malware over several years.

As a second set of experiments, we train DroidApiMiner using a dataset composed of newbenign combined with one of the three most recent malware datasets each (2014, 2015, and 2016). Again, we test DroidApiMiner on all malware datasets. It achieves its

	Testing Sets														
	drebin, oldbenign			2013, oldbenign			2014, oldbenign			2015, oldbenign			2016, oldbenign		
Training Sets	Droid	FAM	MaMa	Droid	FAM	MaMa	Droid	FAM	MaMa	Droid	FAM	MaMa	Droid	FAM	MaMa
drebin,oldbenign	0.32	0.54	**0.96**	0.35	0.50	**0.96**	0.34	0.50	**0.79**	0.30	**0.50**	0.42	0.33	**0.51**	0.43
2013,oldbenign	0.33	0.90	**0.93**	0.36	0.55	**0.97**	0.35	**0.95**	0.74	0.31	**0.87**	0.36	0.33	**0.82**	0.29
2014,oldbenign	0.36	**0.95**	0.92	0.39	**0.99**	0.93	0.62	0.73	**0.95**	0.33	**0.81**	0.79	0.37	**0.82**	0.78
	drebin, newbenign			2013, newbenign			2014, newbenign			2015, newbenign			2016, newbenign		
Training Sets	Droid	FAM	MaMa	Droid	FAM	MaMa	Droid	FAM	MaMa	Droid	FAM	MaMa	Droid	FAM	MaMa
2014,newbenign	0.76	**0.99**	**0.99**	0.75	**0.99**	**0.99**	0.92	0.89	**0.99**	0.67	0.86	**0.89**	0.65	0.82	**0.83**
2015,newbenign	0.68	0.92	**0.98**	0.68	0.84	**0.98**	0.69	0.95	**0.99**	0.77	0.67	**0.95**	0.65	**0.91**	0.90
2016,newbenign	0.33	0.83	**0.97**	0.35	0.69	**0.97**	0.36	0.91	**0.99**	0.34	0.86	**0.93**	0.36	0.53	**0.92**

Table 6.7: F-Measure of MaMaDroid (MaMa) vs our variant using frequency analysis (FAM) vs DroidApiMiner (Droid) [14].

best result when the 2014 and newbenign datasets are used for both testing and training, yielding F-measure of 0.92, which drops to 0.67 and 0.75 one year into the future and past, respectively. With the same datasets, FAM achieves F-measure of 0.89 and it drops to 0.86 one year into the future but rises to 0.99 one year into the past. Comparing DroidApiMiner and FAM shows that DroidApiMiner only performs better than FAM in two out of 15 experiments. In these two experiments, FAM was trained and tested on samples from the same year and resulted in a slightly lower precision, thus, increasing false positives.

As summarized in Table 6.7, MaMaDroid and FAM, respectively, achieves significantly higher performance than DroidApiMiner in all but one and two settings out of 30, showing that abstracting the API calls improves the detection accuracy of our systems. In the settings where DroidApiMiner performs better than MaMaDroid, the malicious training set is much older than the malicious test set.

6.4.5 Runtime Performance

We now analyze the runtime performance of MaMaDroid and FAM, when operating in family, package, or class mode, as well as DroidApiMiner. We run our experiments on a desktop with a 40-core 2.30GHz CPU and 128GB of RAM, but only use 1 core and allocate 16GB of RAM for evaluation.

6.4.5.1 MaMaDroid

We envision MaMaDroid to be integrated in offline detection systems, e.g., run by the app store. Recall that MaMaDroid consists of different phases, so in the following, we review the computational overhead incurred by each of them, aiming to assess the feasibility of real-world deployment.

MaMaDroid's first step involves extracting the call graph from an apk and the complexity of this task varies significantly across apps. On average, it takes 9.2s±14 (min 0.02s, max 13m) to complete for samples in our malware sets. Benign apps usually yield larger call graphs, and the average time to extract them is 25.4s±63 (min 0.06s, max 18m) per app. Next, we measure the time needed to extract call sequences while abstracting to families, packages, or classes depending on MaMaDroid's mode of operation. In family mode, this phase completes in about 1.3s on average (and at most 11.0s) with both benign and malicious samples. Abstracting to packages takes slightly longer, due to the use of 341 packages in MaMaDroid. On average, this extraction takes 1.67s±3.1 for malicious apps and 1.73s±3.2 for benign samples. Recall that in class mode, after abstracting to classes, we cluster the classes to a smaller set of labels due to its size. Therefore, in this mode it takes on average, 5.84s±2.1 and 7.3s±4.2 respectively, to first abstract the calls from malware and benign apps to classes, and 2.74s per app to build the co-occurrence matrix from which we compute the similarity between classes. Finally, clustering and abstracting each call to its corresponding class label takes 2.38s and 3.4s respectively, for malware and benign apps. In total, it takes 10.96s to abstract calls from malware apps to their corresponding class labels and 13.44s for benign apps.

MaMaDroid's third step includes Markov chain modeling and features vector extraction. With malicious samples, it takes on average 0.2s±0.3, 2.5s±3.2, and 1.49s±2.39 (and at most 2.4s, 22.1s, and 46.10s), respectively, with families, packages, and classes, whereas with benign samples, it takes 0.6s±0.3, 6.7s±3.8, and 2.23s±2.74 (at most 1.7s, 18.4s, and 43.98s). Finally, the last step is classification and the performance depends on both the

machine learning algorithm employed and the mode of operation. More specifically, running times are affected by the number of features for the app to be classified, and not by the initial dimension of the call graph, or by whether the app is benign or malicious. Regardless, in family mode, Random Forests, 1-NN, and 3-NN all take less than 0.01s. With packages, it takes, respectively, 0.65s, 1.05s, and 0.007s per app with 1-NN, 3-NN, Random Forests. Whereas it takes, respectively, 1.02s, 1.65s, and 0.05s per app with 1-NN, 3-NN, and Random Forests in class mode.

Overall, when operating in family mode, malware and benign samples take on average, 10.7s and 27.3s respectively, to complete the entire process, from call graph extraction to classification. In package mode, the average completion times for malware and benign samples are 13.37s and 33.83s respectively. Whereas in class mode, the average completion times are respectively, 21.7s and 41.12s for malware and benign apps. In both modes of operation, time is mostly (>80%) spent on call graph extraction.

6.4.5.2 FAM

FAM, which is a variant of MaMaDroid, includes three phases. The first phase, API calls extraction, takes 0.7s±1.5 (min 0.01s, max 28.4s) per app in our malware datasets and 13.2s±22.2 (min 0.01s, max 222s) per benign app. The second phase includes API call abstraction, frequency analysis, and feature extraction. While API call abstraction is dependent on the dataset and the mode of operation, frequency analysis and feature extraction are only dependent on the mode of operation and are very fast in all modes. In particular, it takes on average, 1.32s, 1.69s±3.2, and 5.86s±2.1 respectively, to complete a malware app in family, package, and class modes. Whereas it takes on average, 1.32s±3.1, 1.75s±3.2, and 7.32s±2.1 respectively, for a benign app in family, package, and class modes. The last phase which is classification is very fast regardless of dataset, mode of operation, and classifier used. Specifically, it takes less than 0.01s to classify each app in all modes using the three different classifiers. Overall, it takes in total 2.02s, 2.39s, and

6.56s respectively, to classify a malware app in family, package, and class modes. While with benign apps, the total is 14.52s, 14.95s, and 20.52s respectively, in family, package, and class modes.

6.4.5.3 DroidApiMiner

Finally, we evaluate the runtime performance of DroidApiMiner [14]. Its first step, i.e., extracting API calls, takes 0.7s±1.5 (min 0.01s, max 28.4s) per app in our malware datasets. Whereas it takes on average, 13.2s±22.2 (min 0.01s, max 222s) per benign app. In the second phase, i.e., frequency and data flow analysis, it takes on average, 4.2s per app. Finally, classification using 3-NN is very fast, i.e., 0.002s on average. In total, DroidApiMiner takes respectively, 17.4s and 4.9s for a complete execution on one app from our benign and malware datasets, which while faster than MaMaDroid, achieves significantly lower accuracy. In comparison to MaMaDroid, DroidApiMiner takes 5.8s and 9.9s less on average to analyze and classify a malicious and benign app when MaMaDroid operates in family mode and 8.47s and 16.43s less on average in package mode.

In conclusion, our experiments show that our prototype implementation of MaMaDroid is scalable enough to be deployed. Assuming that, everyday, a number of apps in the order of 10,000 are submitted to Google Play, and using the average execution time of benign samples in family (27.3s), package (33.83s), and class (41.12s) modes, we estimate that it would take less than an hour and a half to complete execution of all apps submitted daily in both modes, with just 64 cores. Note that we could not find accurate statistics reporting the number of apps submitted everyday, but only the total number of apps on Google Play.[10] On average, this number increases by a couple of thousands per day, and although we do not know how many apps are removed, we believe 10,000 apps submitted every day is likely an upper bound.

[10] `http://www.appbrain.com/stats/number-of-android-apps`

6.5 Discussion

Our work yields important insights around the use of API calls in malicious apps, showing that, by abstracting the API calls to higher levels and modeling these abstracted calls, we can obtain high detection accuracy and retain it over several years, which is crucial due to the continuous evolution of the Android ecosystem.

As discussed in Section 6.3, the use of API calls changes over time, and in different ways across malicious and benign samples. From our newer datasets which include samples up to Spring 2016 (API level 23), we observe that newer APIs introduce more packages, classes, and methods, while also deprecating some. Figure 6.7 shows that benign apps use more calls than malicious apps developed around the same time. We also notice an interesting trend in the use of Android (Figure 6.8(a)) and Google (Figure 6.8(b)) APIs: malicious apps follow the same trend as benign apps in the way they adopt certain APIs, but with a delay of some years. This might be a side effect of Android malware authors' tendency to repackage benign apps, adding their malicious functionalities onto them.

Given the frequent changes in the Android framework and the continuous evolution of malware, systems like DroidApiMiner [14] – being dependent on the presence or the use of certain API calls – become increasingly less effective with time. As shown in Table 6.7, malware that uses API calls released after those used by samples in the training set cannot be identified by these systems. On the contrary, as shown in Figure 6.10, MaMaDroid detects malware samples that are *1 year* newer than the training set obtaining an F-measure of 0.86 (as opposed to 0.46 with DroidApiMiner) when the apps are modeled as Markov chains. After 2 years, the value is still at 0.75 (0.42 with DroidApiMiner), dropping to 0.51 after 4 years.

We argue that the effectiveness of MaMaDroid's classification remains relatively high for a longer period, owing to the abstraction of API calls and the Markov models generated from the sequence of API calls. These models tend to be more robust to malware evolution

because abstracting to, e.g., packages makes the system less susceptible to the introduction of new API calls. To verify this, we developed a variant of MaMaDroid named FAM that abstracts API calls and is based on frequency analysis similar to DroidApiMiner. Although, the addition of API call abstraction results in an improvement of the detection accuracy of the system (an F-measure of 0.81 and 0.76 after one and two years respectively), it also resulted in scenarios where there are no API calls that are more frequently used in malware than benign apps.

In general, abstraction allows MaMaDroid capture newer classes/methods added to the API, since these are abstracted to already-known families or packages. As a result, it does not require any change or modification in its operation with newer API releases. In case newer packages are added to newer API level releases, MaMaDroid only requires adding a new state for each new package to the Markov chains, and the probability of a transition from a state to this new state in old apps (i.e., apps without the new packages) would be 0. That is, if only two packages are added in the new API release, only two states need to be added which requires trivial effort. In reality though, methods and classes are more frequently added than packages with new API releases. Hence, we also evaluate whether MaMaDroid still performs as well as in package mode, when we abstract API calls to classes and measure the overall overhead increase. Results from Figure 6.10, and 6.11 indicate that finer-grained abstraction is less effective as time passes when older samples are used for training and newer samples for testing, while they are more effective when samples from the same year or newer than the testing sets are used for training. However, while all three modes of abstraction performs relatively well, we believe abstraction to packages is the most effective as it generally performs better than family – though less lightweight – and as well as class but more efficient.

6.5.1 Case Studies of False Positives and Negatives

The evaluation presented in Section 6.4 shows that our systems detect Android malware with high accuracy. As in any detection system, however, the systems make a small number of incorrect classifications, incurring some false positives and false negatives. Next, we discuss a few case studies aiming to better understand these misclassifications. We focus on MaMaDroid since it achieves better results, and when we experiment with newer datasets, i.e., `2016` and `newbenign` in package mode.

False Positives. First, we analyze the manifest of 164 apps mistakenly detected as malware by MaMaDroid, finding that most of them use "dangerous" permissions [21]. In particular, 67% write to external storage, 32% read the phone state, and 21% access the device's fine location. We further analyzed apps (5%) that use the READ_SMS and SEND_SMS permissions, i.e., even though they are not SMS-related apps, they can read and send SMSs as part of the services they provide to users. In particular, a *"in case of emergency"* app is able to send messages to several contacts from its database (possibly added by the user), which is a typical behavior of Android malware in our dataset, ultimately leading MaMaDroid to flag it as malicious. As there are sometimes legitimate reasons for benign apps to use permissions considered to be dangerous, we also analyze the false positives using a second approach. Specifically, we examine the average number of the 100 most important features used by MaMaDroid to distinguish malware from benign apps that are present in the false positive samples. We select the top 100 features as it represents no more than about 4.5% of the features (there are 2202 features in the `2016` and `newbenign` datasets). As we show in Figure 6.15, for 90% of the false positives, there are only on average, 33 of the most important features present in their features vectors which is similar to the behavior observed in the true positives (34/100). MaMaDroid misclassifies these samples because they exhibit similar behavior to the true positives and these samples could be further manually analyzed to ascertain maliciousness.

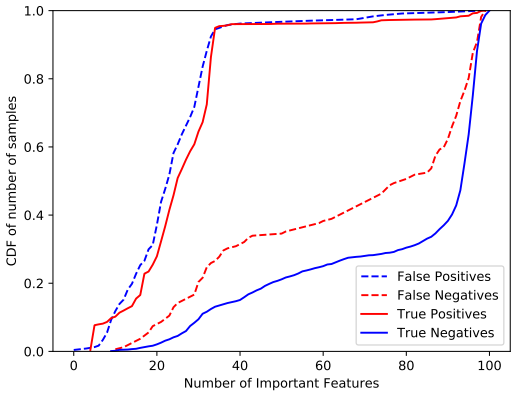


Figure 6.15: CDF of the number of important features in each classification type.

False Negatives. We also check 114 malware samples missed by MaMaDroid using VirusTotal.[11] We find that 18% of the false negatives are actually not classified as malware by any of the antivirus engines used by VirusTotal, suggesting that these are actually legitimate apps mistakenly included in the VirusShare dataset. 45% of MaMaDroid's false negatives are *adware*, typically, repackaged apps in which the advertisement library has been substituted with a third-party one to create monetary profit for the developers. Since they are not performing any clear malicious activity, MaMaDroid is unable to identify them as malware. Finally, we find that 16% of the false negatives reported by MaMaDroid are samples sending text messages or starting calls to premium services. We also do a similar analysis of false negatives with similar results. That is, there a few more adware samples (53%), but similar percentages for potentially benign apps (15%) and samples sending SMSs or placing calls (11%). We also investigate further, the false negatives using the 100 most important features of MaMaDroid. We find that for 90% of the false negatives, they have on average, 97 of the 100 features similar to the true negatives (98/100).

[11] `https://www.virustotal.com`

6.5.2 Evasion

Next, we discuss possible evasion techniques and how they can be addressed. One straightforward evasion approach could be to repackage a benign app with small snippets of malicious code added to a few classes. However, it is difficult to embed malicious code in such a way that, at the same time, the resulting Markov chain looks similar to a benign one. For instance, the running example from Section 6.2 (malware posing as a memory booster app and executing unwanted commands as root) is correctly classified by MaMaDroid; although most functionalities in this malware are the same as the original app, injected API calls generate some transitions in the Markov chain that are not typical of benign samples.

The opposite procedure, i.e., embedding portions of benign code into a malicious app, is also likely ineffective against MaMaDroid, since, for each app, we derive the feature vector from the transition probability between calls over the entire app. A malware developer would have to embed benign code inside the malware in such a way that the overall sequence of calls yields similar transition probabilities as those in a benign app, but this is difficult to achieve because if the sequences of calls have to be different (otherwise there would be no attack), then the models will also be different.

Moreover, attackers could try using reflection, dynamic code loading, or native code [167]. Because MaMaDroid uses static analysis, it fails to detect malicious code when it is loaded or determined at runtime. However, MaMaDroid can detect reflection when a method from the reflection package (`java.lang.reflect`) is executed. Therefore, we obtain the correct sequence of calls up to the invocation of the reflection call, which may be sufficient to distinguish between malware and benign apps. Similarly, MaMaDroid can detect the usage of class loaders and package contexts that can be used to load arbitrary code, but it is not able to model the code loaded; likewise, native code that is part of the app cannot be modeled, as it is not Java and is not processed by Soot. These limitations are not specific to MaMaDroid, but common to static analysis in general, and could be possibly mitigated

using MaMaDroid alongside dynamic analysis techniques.

Another approach could be using dynamic dispatch so that a class X in package A is created to extend class Y in package B with static analysis reporting a call to root() defined in Y as X.root(), whereas at runtime Y.root() is executed. This can be addressed, however, with a small increase in MaMaDroid's computational cost, by keeping track of self-defined classes that extend or implement classes in the recognized APIs, and abstract polymorphic functions of this self-defined class to the corresponding recognized package, while, at the same time, abstracting as self-defined overridden functions in the class.

Finally, identifier mangling and other forms of obfuscation could be used aiming to obfuscate code and hide malicious actions. However, since classes in the Android framework cannot be obfuscated by obfuscation tools, malware developers can only do so for self-defined classes. MaMaDroid labels obfuscated calls as `obfuscated` so, ultimately, these would be captured in the behavioral model (and the Markov chain) for the app. In our sample, we observe that benign apps use significantly less obfuscation than malicious apps, indicating that obfuscating a significant number of classes is not a good evasion strategy since this would likely make the sample more easily identifiable as malicious. Malware developers might also attempt to evade MaMaDroid by naming their self-defined packages in such a way that they look similar to that of the `android` or `google` APIs, e.g., java.lang.reflect.*malware* is easily prevented by first abstracting to classes before abstracting to any further modes as we already do in Section 6.2.1.2.

6.5.3 Possible Bias

The performance of MaMaDroid may suffer from possible bias as recently highlighted by Pendlebury et al. [163]. Here, we provide a review of the biases and whether MaMaDroid suffer from them.

Pendelebury et al. [163] presents Tesseract, which attempts to eliminate spacial and temporal bias present in malware classifiers. To eliminate these biases, the authors suggest

malware classifiers should enforce three constraints: *Temporal training consistency* i.e., all objects in the training set must temporally precede all objects in the testing set, *Temporal goodware/malware windows consistency* i.e., in every testing slot of size Δ, all test objects must be from the same time window, and *Realistic malware-to-goodware percentage in testing* i.e., the testing distribution must reflect the real-world percentage of malware observed in the wild.

With respect to the constraints that eliminate temporal bias, recall that we evaluate MaMaDroid over several experimental settings and some of these settings do not violate these constraints. For example, there is temporal goodware/malware windows consistency in many of the settings when it is evaluated using samples from the same year (e.g., newbenign and 2016) and when it is trained on older (resp. newer) samples and tested on newer (resp. older) samples e.g., oldbenign and 2013. While temporal training consistency shows the performance of the classifier in detecting unknown samples (especially when the unknown samples are not derivatives of previously known malware family), many malware classifiers usually have false negatives. Malware families in these false negatives could form the base of present or "future" malware. For example, samples previously classed as goodware from Google Play Store are from time to time detected as malware [48, 49, 197, 207]. As a result of samples previously detected as goodware now being detected as malware, we argue that robust malware classifiers should be able to detect previous (training with newer samples and testing on older samples), present (training and testing on samples from the same time window), and future (training on older samples and testing on newer samples) malware objects effectively.

During the evaluation of MaMaDroid, we do not enforce the constraint that eliminates spacial bias as proposed in Tesseract. When we evaluate MaMaDroid, the minimum and maximum percentages of malware in the testing set are 49.7% and 84.85%, respectively, which effects the Precision and Recall. But as already highlighted by Pendelebury et al. [163], estimating the percentage of malicious Android apps in the wild with respect to

goodware, is a non-trivial task, with different sources reporting different results [105, 135].

6.5.4 Limitations

MaMaDroid requires a sizable amount of memory in order to perform classification, when operating in package mode, working on more than 100,000 features per sample. The quantity of features, however, can be further reduced using feature selection algorithms such as PCA. As explained in Section 6.4, when we use 10 components from the PCA, the system performs almost as well as the one using all the features; moreover, using PCA comes with a much lower memory complexity in order to run the machine learning algorithms, because the number of dimensions of the features space is remarkably reduced.

Soot [202], which we use to extract call graphs, fails to analyze some apks. In fact, we were not able to extract call graphs for a fraction (4.6%) of the apps in the original datasets due to scripts either failing to apply the `jb` phase, which is used to transform Java bytecode to the primary intermediate representation (i.e., jimple) of Soot or not able to open the apk. Even though this does not really affect the results of our evaluation, one could avoid it by using a different/custom intermediate representation for the analysis or use different tools to extract the call graphs which we plan to do as part of future work.

In general, static analysis methodologies for malware detection on Android could fail to capture the runtime environment context, code that is executed more frequently, or other effects stemming from user input [25]. These limitations can be addressed using dynamic analysis, or by recording function calls on a device. Dynamic analysis observes the live performance of the samples, recording what activity is actually performed at runtime. Through dynamic analysis, it is also possible to provide inputs to the app and then analyze the reaction of the app to these inputs, going beyond static analysis limits. To this end, in the next chapter we employ MaMaDroid in a dynamic analysis setting and investigate how it performs compared to static analysis.

6.5.5 Concluding Remarks

In this chapter, we presented MaMaDroid for detecting malware statically by abstracting the sequence of API calls; then, modeling the API calls as Markov chains; and finally, using the transition probabilities of the Markov chains as the features that identify an app. We showed that these techniques allow our detection tools to detect malware at high accuracy even after one year. Compared to DroidAPIMiner, the detection performance of MaMaDroid is less susceptible to malware and the Android framework evolution. Our results emphasized the importance of designing detection systems that do not easily go out-of-date when malware evolve their evasion techniques. The continuous evolution of malware evasion techniques result in an arms race, with the malware developers evolving their techniques when their apps start being detected. Although the performance of MaMaDroid also deteriorates after two years (which may be due to the characteristics of our dataset i.e., malware lagging benign apps by upto two years in the way they use API calls, which is mentioned in Section 6.4.1.2), it could serve as a building block for more robust designs by the research community. Finally, we make the code of MaMaDroid as well as the hash of the samples in our datasets publicly available[12] and, on request, the apk samples, parsed call graphs, and abstracted sequences of API calls on which MaMaDroid has been evaluated on.

[12] `https://bitbucket.org/gianluca_students/mamadroid_code`

Chapter 7

A Family of Droids: Analyzing Behavioral Model-based Android Malware Detection via Static and Dynamic Analysis

---

In this chapter, we address **RQ4**: Does having humans test apps during dynamic analysis improve malware detection compared to pseudorandom input generators such as Monkey [107]? and **RQ5**: How do different analysis methods (i.e., static, dynamic, and hybrid analysis) compare to each other, in terms of malware detection performance when the same technique is used to build the detection models?

The work presented in this chapter has been published in PST [156]

7.1 Motivation

7.1.1 The Research Problem

As the Android framework and malware that targets it evolve, so do the techniques that detect these malware; the techniques either undergo modifications or change completely. The research community has proposed a number of techniques to detect and block malware based on either static or dynamic analysis. With the former, the code is recovered from the apk, and features are extracted to train machine learning classifiers; with the latter, apps are executed in a controlled environment, usually on an emulator or a virtual device, via a real person or an automatic input generator such as Monkey [107].

In particular, a few approaches have been recently proposed aiming to improve accuracy

of malware detection. *(1) Behavioral Modeling:* MaMaDroid which is presented in the previous chapter, uses static analysis to build a behavioral model of malware samples, relying on the *sequences* of abstracted API calls; this yields high accuracy and is also resilient to API changes, thus, reducing the need to re-train models. *(2) Input Generators:* previous studies [20,46,140] have introduced input generators that aim to mimic app usage by humans, more effectively than the standard Android pseudorandom input generator (Monkey), thus improving the chances of triggering malicious code during execution. *(3) Hybrid Analysis:* by combining static and dynamic analysis, hybrid analysis has been used to get the best of the two worlds, typically, following two possible strategies. One approach is to use static analysis to gather information about the apps under analysis (e.g., intent filters an app listens for, execution paths to specific API calls, etc.) and then execute apps in the dynamic analysis stage, while ensuring that all execution paths of interest are triggered [46, 212]; in the other approach, features extracted using static analysis (e.g., permissions, API calls, etc.) are combined with those from dynamic analysis (e.g., file access, networking events, etc.), and used to train an ensemble machine learning model [133,134].

Overall, despite a large body of work proposing various Android malware detection tools, the research community's stance on whether to use static or dynamic analysis primarily stems from the systems' limitations and the vulnerabilities to possible evasion techniques faced by each approach. For instance, static analysis methods that extract features from permissions requested by apps often yield high false positive rates, since benign apps may actually need to request permissions classified as dangerous [77], while systems that perform classification based on the frequency of API calls [14] often require constant retraining; moreover, reflection and dynamic code loading can be used to evade static analysis-based detection. On the other hand, while dynamic analysis tools are not susceptible to the above evasion techniques, their accuracy is greatly dependent on whether malicious code is actually triggered during test execution, and in general dynamic analysis often does not

scale. Nonetheless, we still lack a deep understanding of whether detection techniques implemented in one approach (e.g., static) may actually perform better if implemented in the other approach (dynamic).

In this chapter, we aim to fill this research gap by addressing RQ4 and RQ5, summarized here as:

- Will having humans test apps during dynamic analysis, improve malware detection?
- How do different analysis methods (i.e., static, dynamic, and hybrid analysis) compare to each other, when the same underlying detection technique is employed?

7.1.2 The Research Roadmap

In order to address RQ4, we modify CHIMP [17], a platform that allows the crowdsourcing of human inputs for testing Android apps, to support building a behavioral model-based malware detection system (as per MAMADROID). This modification allows us extract sequences of abstracted API calls from the traces produced while executing an app in a virtual device (instead of extracting the calls from the apk as per MAMADROID). We call the resulting system AUNTIEDROID. We then instantiate AUNTIEDROID using both an automated pseudorandom input generator (i.e., Monkey) and user-generated inputs and evaluate how they effect its detection performance. To address RQ5, we first select the technique we introduced in MAMADROID (see Chapter 6) as the underlying detection technique across all analysis methods. Because as we showed earlier, it effectively detects malware at a high accuracy even after one year. We implement MAMADROID's technique in a dynamic analysis setting as AUNTIEDROID, and to obtain a hybrid version, we merged the sequence of abstracted API calls from static and dynamic analysis.

The work presented in this chapter includes the contributions of the other coauthors as outlined in Chapter 1.4. In particular, the modification of the virtual device (Section 7.2.1), app stimulation integration (Section 7.2.2), and the comparative analysis of the misclassi-

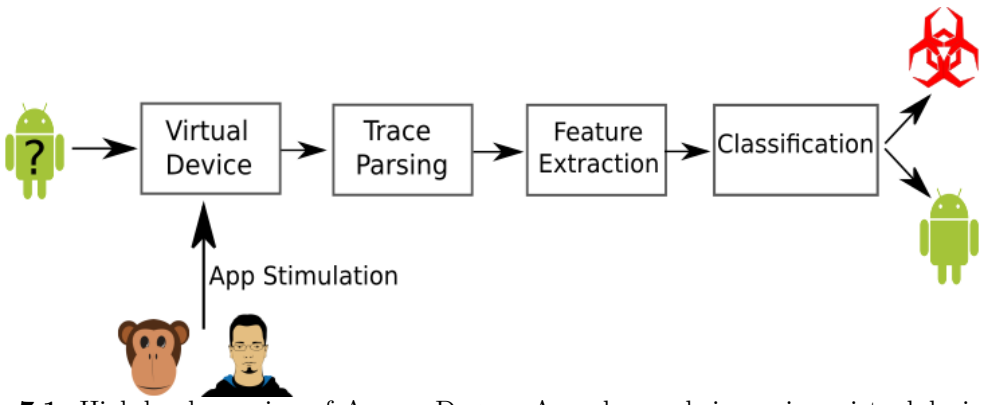


Figure 7.1: High-level overview of AuntieDroid. An apk sample is run in a virtual device, using either Monkey or human. Then, the APIs called during execution are parsed and used for feature extraction. Finally, the app is classified as either benign or malicious.

fication across analysis methods (Section 7.5.2.2) are the contributions of coauthors.

The rest of this chapter is structured as follows: Section 7.2 presents the design and implementation of AuntieDroid, while Section 7.3 presents the experiments carried out as well as the datasets used for the study. Section 7.4 presents the results obtained when each analysis method is evaluated and Section 7.5 shows how they compare to each other. Finally, Section 7.6 discusses the challenges as well as the limitations of our work.

7.2 AuntieDroid: Overview and Implementation

In this section, we present AuntieDroid, a system performing Android malware detection based on behavioral models extracted through dynamic analysis. Our main objective is to compare its performance to its static analysis counterpart, i.e., MaMaDroid (see Chapter 6). In fact, we build on it, in that we again model the sequences of (abstracted) calls as Markov chains, and use the transition probabilities between states as features.

In order to build the behavioral model in dynamic analysis, we modify a virtual device to allow us to capture the sequence of API calls from the runtime execution trace of apps. We call the resulting system AuntieDroid, and summarize its operation in Figure 7.1. First, we execute apps in a virtual device, stimulated by either an automated program

(Monkey) or a human. We then parse the traces generated by the executions, and extract features for classification.

7.2.1 Virtual Device

As mentioned above, the first step in AuntieDroid is to execute apk samples in a virtual device with either (i) human users or (ii) an UI automation tool like the Monkey [107]. Our virtual device testbed, described in detail below, builds on Chimp, an Android testing system recently presented by Almeida et al. [17] which can be used to collect human inputs from mobile apps.

Chimp [17]. Chimp virtualizes Android devices using the Android-x86 platform, running behind a QEMU instance on a Linux server. Although it uses an x86 Android image, Chimp actually supports two application binary interfaces (ABI), i.e., both ARM and x86 instruction sets are supported. Once running, the virtualized device can be stimulated by either a locally running automated tool (e.g., Monkey), or the UI can be streamed to a remote browser, allowing humans to interact with it. Chimp can be used to collect a wide range of data (user interactions, network traffic, performance, etc.) as well as explicit user feedback; however, for the sake of AuntieDroid, we modify it to generate and collect *run-time traces*, i.e., the call graph of an app's interactive execution.

Modifications to Chimp. In order to effectively monitor malware execution, we substantially modify the original prototype of Chimp, which was primarily designed to enable large-scale, human testing of benign apps. In fact, the original prototype supports code instrumentation via a Java code coverage library called EMMA [75], unfortunately, EMMA requires an app's source code to be instrumented, which is often not accessible for closed-source apps such as those analyzed in our work. Therefore, we modify Chimp to get access to debug level run-time information from un-instrumented code. Note that in Android, each app runs on a dedicated VM which opens a debugger port using Java's Debug Wire Protocol (JWDP). As long as the *device* is set as debuggable (`ro.debuggable` property),

we can connect to the VM's JWDP port to activate VM level *method tracing*.

We also have to activate tracing: in Android, one can either use Android's Activity Manager (AM) or the DDM Service. Both end up enabling the same functionality – i.e., `startMethodTracing` on `dalvik.system.VMDebug` and `android.os.Debug` – but through different approaches. That is, AM (via `adb am`) exposes a limited API that eventually reaches the app via Inter-Process Communication (IPC), while the DDM Service (DDMS, as used by Android Studio) opens a connection directly to the VM's debugger, providing fine grain control over tracing parameters. We choose the second approach since it is parameterizable, allowing us to set the trace buffer size, which by default (8MB) can only hold a few seconds of method traces. Hence, we implement a new DDM Service in CHIMP, using the ddmlib library [69] to communicate with the VMs and activate tracing. Our DDM service multiplexes all tracing requests through a single debugger and we further modify the ddmlib tracing methods to dump traces to the VM file system, and set the trace buffer size to 128MB. However, apps tested on the virtual device can generate more than 128MB of traces, thus, we add a background job that retrieves and removes traces from the VMs every 30s. Besides preventing the tracing buffer from filling up, this lets us capture partial traces for apps that might crash during stimulation.

To deal with malware masquerading as legitimate applications by using the same package names, we uniquely identify apps based on a hash of the binary, salted with the package name and a unique testing campaign identifier. This is then mapped to the app's storage location on disk. We also modify CHIMP to disable the app verification security feature in Android that attempts to prevent the installation of malicious apps.

7.2.2 App Stimulation

As mentioned, to stimulate the Application Under Analysis (AUA), we use both Monkey and humans.

Monkey [107]. Monkey is Android's de-facto standard UI automation tool used to generate

inputs. In AuntieDroid, "Monkeys" (i.e., more than one Monkey instance) are deployed on the same machine that the virtual devices are running on. We set Monkeys to run a single app for 5 minutes (one virtual device VM per app): each Monkey is setup to generate events every 100ms and ignore timeouts, crashes, and security exceptions (although we still log and process them). Setting Monkey to generate input for 5 minutes only does not adversely affect code coverage, as prior work [58] reports that most input generators achieve maximum coverage between 5 to 10 minutes. As Monkey may generate events at a higher frequency than some apps can process (e.g., generating ANR events), we also re-run offending apps with a decreased rate (300ms).

Humans. In order to have real users stimulate the samples, we recruit about 5k (5,030) workers from the Crowdflower.com crowdsourcing platform. We let them interact with the virtual device by streaming its UI to their browser via an HTML5 client. The client transmits user actions back to AuntieDroid, which translates them to Android inputs and forwards them to the virtual device. In addition to the virtual device UI, user controls were provided to, e.g., move to the next app in the testing session.

Each user is given 4 randomly selected apps from our dataset and told to explore as much of its functionality as possible before proceeding to the next app. We do not enforce a lower bound on the time users must spend testing apps, since given the nature of our sample, some apps might have limited interaction opportunities. Consequently, we let at least three different users stimulate the same app. We also limit app install time to 40s to avoid frustrating the users and we pay each user $0.12 per session (i.e., a single test of session of the 4 randomly selected apps), with the test sessions running from August 9th to 11th, 2017.

Ethics. For the experiments involving humans, we have obtained approval through our institution's ethical review process. Although we requested basic demographic data (age, gender, country), we did not collect privacy sensitive information, and users were instructed not to enter any real, personal information, e.g., account details. Participants did not have

to install any software on their devices: they interacted with the apps on a webpage, while we collected information about the app execution in the background. Also note that we provided email credentials to use when required, so that they did not have to use their own credentials or other contact information.

7.2.3 Trace Parsing

As discussed above, our virtual device component takes care of collecting method traces, network packets, and event logs generated when the app is running. To parse these traces, one could use different strategies, for instance, tracking data flow from selected sources (e.g., the device id – `getDeviceID()`) to sinks (e.g., a data output stream – `writeBytes()`), or using frequency analysis to derive commonly used API calls by malware (as in DroidAPIMiner [14]).

AuntieDroid follows the behavioral model based approach of MaMaDroid, but it is based on the *sequences* of API calls that the app performs at runtime, rather than statically extracting it from the apk. This way, we aim to capture different behavior when benign and malicious apps invoke API calls that access sensitive information. For instance, a benign SMS app might receive an SMS, get the message body using `getMessageBody()` and afterwards, display the message to a user via a view by executing, in sequence, `setText(String msg)` and `show()` methods of the view. A malicious app, however, might exfiltrate all received SMSs by executing `sendTextMessage()` for every message before displaying it.

To derive the API call sequences, we collect the method traces and transform them into a call graph using *dmtracedump* [170]. From the call graph, we then extract the sequences using a custom script, while preserving the number of times an API call is executed as a multiplier in each sequence. As discussed above, to avoid losing traces when the trace buffer is full, we collect virtual device traces every 30s, and clear the buffer for incoming traces. Along the same lines, we have a median of three different users run the same app to improve the quality of the traces gathered. As a result, we aggregate the sequences of API calls they

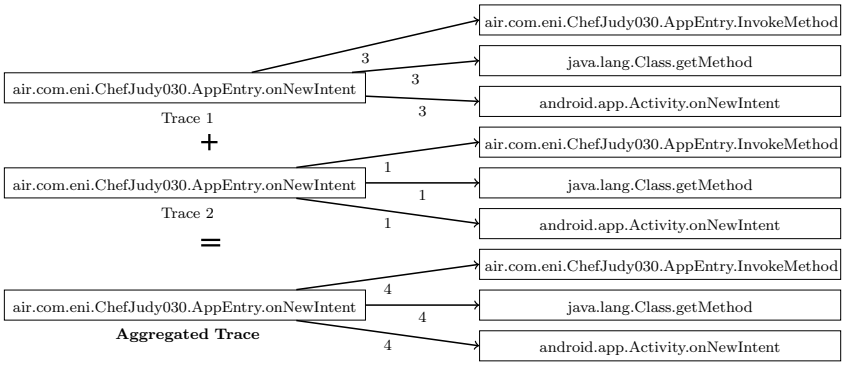


Figure 7.2: Aggregated sequence of API calls showing the direct children of the call `air.com.eni.ChefJudy030.AppEntry. onNewIntent`, along with the number of times they are called (numbers on the arrow).

generate for the same app into a single sequence. In Figure 7.2, we provide an example of the sequence for the API call `air.com.eni.ChefJudy030.AppEntry.onNewIntent` when aggregated from two other sequences. We do not show the params and return type to ease presentation. Also, in some cases, *Trace 1* may contain API calls in a sequence that is not called in *Trace 2*, hence, the aggregated trace also reflects such calls.

7.2.4 Feature Extraction and Classification

As in MaMaDroid, which operates in one of three modes i.e., family, package, or class, AuntieDroid also abstracts each API call in the parsed trace to its corresponding family, package or class names using the Android API packages from API level 26 and the latest Google API packages. The abstraction allows for resilience to API changes in the Android framework as packages are added or deprecated less frequently compared to single API calls. It also helps to reduce the feature set size as the feature vector of each app is the square of the number of states in the Markov chain.

We then build a *behavioral model* from the abstracted sequences of API calls by building a Markov chain that models the sequences from which we extract features used to classify an app as either benign or malicious. More specifically, features are the probability of transitioning from one state (i.e., API call) to another in the Markov chain representing

the API call sequences in an app.

Classification. Finally, we perform classification – i.e., predicting an app as benign or malware, using a supervised machine learning classifier. More specifically, we use Random Forests with 10-fold cross validation. We choose Random Forests since it performs well on binary classification over high-dimension feature spaces, which is the case with the Markov chain based model from MaMaDroid.

7.3 Experimental Setup

In this section, we introduce our experiments, the datasets used in our evaluation, as well as the preprocessing carried out on them.

7.3.1 Overview of Experiments and Detection Approaches

As discussed in Section 7.1, we aim to perform a comparative analysis of Android malware detection systems based on behavioral models, using static, dynamic, and hybrid analysis. To this end, we perform three sets of experiments. (1) *Static:* We evaluate MaMaDroid, which performs Android malware detection based on behavioral modeling in static analysis; (2) *Dynamic:* We analyze the detection performance of AuntieDroid, comparing both automated input generation (Monkey) and human-generated input; (3) *Hybrid:* We combine static and dynamic analysis by merging the sequences of API calls from both methods, once again comparing Monkey and human based input generation.

All methods operate in one of two levels of abstraction, i.e., API calls are abstracted to either their family or package names. We do not evaluate the performance of the system in class mode since as shown in the previous chapter (see Section 6.4.1), abstracting to class does not provide a significant increase in detection accuracy over package mode. Overall, we use the same modeling technique and the same machine learning classifier. More specifically, we use the Random Forests classifier and in `family` (resp., `package`)

mode, we use a configuration of 51 (resp., 101) trees with depth equal to 8 (resp., 32).

7.3.2 Datasets

Our evaluation uses two datasets: a collection of recent malware samples and a dataset of random benign apps, as discussed below.

Benign Samples. For consistency, we opt to re-use the set of 2,568 benign apps labeled as "newbenign" in the previous chapter. In June 2017, we re-downloaded all the apps in order to ensure we have working apps and their latest version, obtaining 2,242 (87%) apps. We complement this list with a 33% sample of the top 49 apps (as of June 2017) from the 29 categories listed on the Google Play Store, adding an additional 481 samples. Overall, our benign dataset includes a total of 2,723 apps.

Malware Samples. Our malware dataset includes samples obtained in June 2017 from VirusShare – a repository of apps that are likely to be malicious.[1] More precisely, VirusShare contains samples that have been detected as malware on various OS platforms, including Android. To obtain only Android malware, we check that each sample is correctly zipped and packaged as an apk, contains a Manifest file, and has a package name. Using this method, we gather 2,692 valid Android samples labeled as malware in 2017 by the anti-virus engines on VirusShare. In addition, we add two more apps (Chef Judy and Dress Up Musa) from the Google Play Store reported as malware in the news and later removed from the play store.[2] In total, our malware dataset includes 2,694 apps.

7.3.3 Dataset Preprocessing

In order to evaluate each detection method, we first pre-process the samples in our dataset as described below.

Static Analysis. For static analysis, we re-use MaMaDroid and set a timeout limit of

[1] `https://virusshare.com/`

[2] See `https://goo.gl/hBjm0T` and `https://goo.gl/IQprtP`

Failure	Benign	Malware
Already installed	10	9⋆
Contains native code not compatible with the device's CPU	0	2
App's dex files could not be optimized and validated	0	1
Apk could not be unarchived by Android aapt	3	4
Shared library requested by app is not available on the device	0	1
Does not support the SDK (version 4.4.2) on the device	36	6
Requests a shared user already installed on the device	0	1
Android's failure to parse the app's certificate	0	4
Fails to complete installation within time limit (40s)	39	23
Total:	**88**	**51**

Table 7.1: Reasons why apps fail to install on the virtual device. ⋆These are repackaged Google apps i.e., they have the same package names as the Google apps but with a different hash.

six hours during the call graph extraction phase. We could not obtain the call graphs for 98 (3.6%) and 251 (9.3%) apps in the benign and malware datasets, respectively. This is consistent with experiments reported in Chapter 6 and it is due to the timeout and samples exceeding memory requirement (we allocate 16GB for the JVM heap space).

Dynamic Analysis. During dynamic analysis, before running the apps on the virtual device, we process them statically using Androguard[3] to determine whether they have activities. Out of the total 5,417 apps in our datasets, we find that 82 apps contain no activity. As interaction with an Android app requires visuals that users can click, tap, or touch to trigger events, we therefore exclude these from the samples to be stimulated using Monkey or humans. We also remove 244 apps that do not have a launcher activity, since launching these apps on the virtual device will have no visual effect; i.e., no UI will be displayed to the tester. Finally, we do not include 139 apps which fail to install on the virtual device for one of the reasons shown in Table 7.1.

Hybrid Analysis. To obtain a hybrid detection system, we merge the sequences of abstracted API calls (from which we extract features) obtained using both static and dynamic analysis. More specifically, we merge the sequences of API calls following the same strategy

[3] `https://github.com/androguard/androguard`

Analysis	Stimulator	Category	#Samples	#Traces / Call graphs
Static (MaMaDroid)	–	Benign	2,723	2,625
		Malware	2,694	2,443
Dynamic (AuntieDroid)	Human	Benign	2,596	2,348
		Malware	2,356	1,911
	Monkey	Benign	2,596	2,336
		Malware	2,356	1,892
Hybrid	Static & Human	Benign	2,596	2,235
		Malware	2,356	1,708
	Static & Monkey	Benign	2,596	2,234
		Malware	2,356	1,686

Table 7.2: Datasets used to evaluate each method of analysis.

used to aggregate the traces discussed in Section 7.2.3. Naturally, for hybrid analysis, we use samples for which we have traces for both static and dynamic analysis.

Final Datasets. In Table 7.2 we report, in the right-most column, the final number of samples in each dataset, for each method of analysis. During dynamic analysis, we fail to obtain traces for 724 apps when stimulating with Monkey and 693 when stimulating with humans. We discuss the reasons for the failure after examining the logs in Section 7.6.1. Note that the hybrid analysis method consists of samples for which we obtain traces both statically and dynamically.

7.4 Evaluation of Detection Approaches

In this section, we present the results of our experimental evaluation, reporting both detection performance and, for dynamic analysis, code coverage. We compare the detection performance of each approach in terms of F-measure as it is the harmonic mean of the Recall and Precision of the classification model. In scenarios where the prediction of a class (e.g., the malware class) is of utmost importance, Precision can be used. As a result,

Analysis	Stimulator	Mode	F-measure	Precision	Recall
Static	–	`Family`	**0.86**	0.84	0.88
(MaMaDroid)		`Package`	**0.91**	0.89	0.93
Dynamic	Human	`Family`	**0.85**	0.80	0.90
(AuntieDroid)		`Package`	**0.88**	0.84	0.92
	Monkey	`Family`	**0.86**	0.84	0.89
		`Package`	**0.92**	0.91	0.93
Hybrid	Static & Human	`Family`	**0.87**	0.86	0.88
		`Package`	**0.90**	0.88	0.91
	Static & Monkey	`Family`	**0.88**	0.88	0.89
		`Package`	**0.92**	0.92	0.93

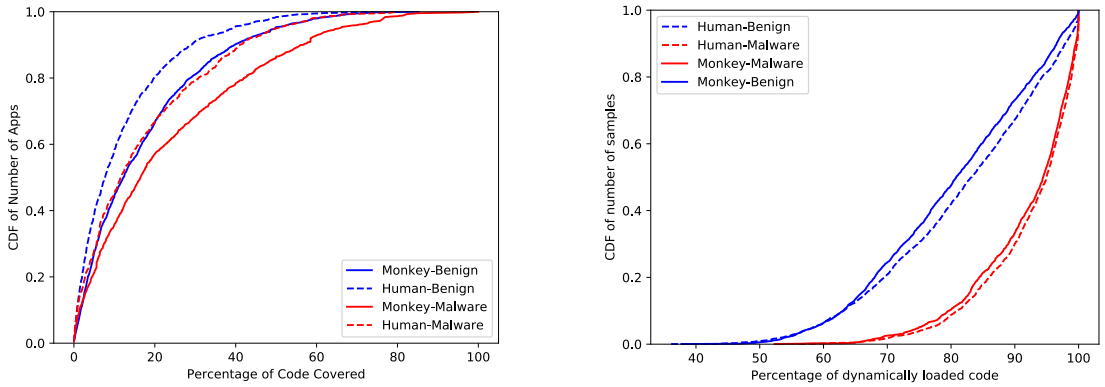
Table 7.3: Results achieved by all analysis methods while using human and Monkey as app stimulators during dynamic analysis.

we report along with the F-measure, the Precision and Recall for each approach.

7.4.1 Static Analysis

In static analysis, we train and test the family and package modes of MaMaDroid using the dataset presented in Table 7.2. Note that while MaMaDroid as presented in the previous chapter uses API level 24, we use API level 26 (the most recent at the time of this study) in this chapter.

We run our experiments on the samples (2,625 benign and 2,443 malware) for which we obtain call graphs, and report the F-measure obtained when operating in family and package modes in the top two rows of Table 7.3. We observe that the latter performs slightly better, achieving F-measure of 0.91, compared to 0.86 in the former which is consistent with the results reported in Chapter 6. The package mode achieves higher F-measure than family mode as it captures the behavior of apps at a finer granularity which reveals more distinguishing behaviors between malware and benign apps as demonstrated by higher Precision and Recall (see Table 7.3).



(a) (b)

Figure 7.3: CDF of the percentage of (a) code covered in benign and malicious apps when they are stimulated by Monkey and human, and (b) API calls that are dynamically loaded during dynamic analysis.

7.4.2 Dynamic Analysis

Next, we report the results achieved by dynamic analysis (i.e., using AuntieDroid), comparing between stimulation performed by Monkey and humans.

Detection Performance. For Monkey, we evaluate the classifier on the dataset shown in Table 7.2, i.e., on 2,336/1,892 samples for benign/malware. When AuntieDroid runs in family mode, it achieves F-measure, precision, and recall of 0.86, 0.84, and 0.89, respectively. Whereas in package mode, it achieves F-measure, precision, and recall of 0.92, 0.91, and 0.93, respectively, as reported in Table 7.3. When humans stimulate the apps (2,348 benign and 1,911 malware) and AuntieDroid runs in family mode, the F-measure, precision, and recall are 0.85, 0.80, and 0.90, respectively. Whereas when operating in package mode, F-measure, precision, and recall go up to 0.88, 0.84, and 0.92, respectively (see Table 7.3).

Overall, lower F-measures in all modes of operation in dynamic analysis compared to static analysis (i.e., AuntieDroid vs MaMaDroid) are due to increases in false positives. In fact, recall is around 0.90 on all experiments, while precision is as low as 0.80 (family mode with humans).

Code Coverage. As mentioned earlier, the performances of dynamic analysis tools are affected by whether malicious code is triggered during execution. Since AuntieDroid relies on the sequences of API calls to detect malware, we analyze code coverage of each app to measure how much of an app's API calls Monkey/humans successfully trigger. To this end, we focus on API calls that begin with the package name of the app.

For the apps for which we obtain traces (85% and 86%, resp., for Monkey and human), Monkey is able to trigger on average, 20% of the API calls. In Figure 7.3(a), we plot the cumulative distribution function (CDF) of the percentage of code covered, showing that for 90% of the benign apps, at least 40% of the API calls are triggered by Monkey. Whereas with respect to the malware samples, at least 57% of the API calls are triggered. As for humans, we find that users are able to trigger, on average, 14% of the API calls. Similarly, Figure 7.3(a) shows that at least 29% of the API calls are triggered in 90% of the benign apps. However, with 90% of the malicious apps, 41% of the API calls are triggered.

With both stimulators, there is a higher percentage of code coverage in the malware apps than in benign apps. This is due to malware apps being smaller in size compared to the benign apps in our dataset. The mean number of API calls in the benign and malware apps are respectively, 43,518 and 16,780. However, with respect to stimulators, Monkey is able to trigger more code in apps compared to humans, which is likely due to Monkey triggering more events (at a rate of 100ms for 5 mins) than humans in the time each spend testing the apps.

Dynamic Code Loading. We also report the percentage of code that is dynamically loaded by apps by examining the API calls that exist in the dynamic traces but not in the apk. That is, we extract the API calls from the apk using apktool [23] and measure the percentage of the calls that are in the dynamic traces but not in that extracted statically. We do this to evaluate the prevalence of dynamic code loading in apps as this could be used for malicious purposes [167]. For instance, malware developers may design their apps (or repackage benign apps) to only dynamically load the malicious code and evade static

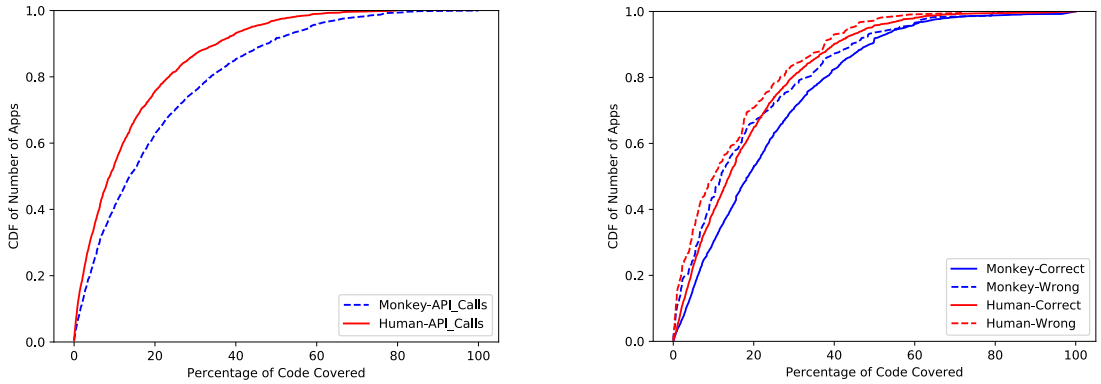
analysis tools [191, 226]. In Figure 7.3(b), we report the CDF of the percentage of the code that is dynamically loaded, finding that in 90% of the samples, 97% of the API calls are dynamically loaded irrespective of the app stimulator. As for malware and benign apps, about 99.5% of the API calls are dynamically loaded in 90% of malware samples irrespective of the stimulator compared to about 97% in benign apps. We also evaluate the percentage of the dynamically loaded code that is common to all apps to measure whether they are primarily virtual device or OS operations such as app installation, start/stop method tracing, os handler etc. We find that only 0.14% and 0.08%, respectively, are common across apps when the stimulator is human and Monkey.

Overall, with up to 99.5% of code being dynamically loaded in 90% of our malware samples, we believe that dynamic code loading might indeed pose a problem for malware detection tools based solely on static analysis and dependent on API calls. That is, these tools might not be resilient to evasion techniques that load code dynamically, since a large portion of the executed code will not be examined at all. On the other hand, with benign apps also heavily employing dynamic code loading, we cannot conclude that only apps that load a large portion of their code during dynamic analysis are malicious.

7.4.3 Hybrid Analysis

We now report the results achieved by hybrid analysis, comparing between stimulation performed by Monkey and humans. Recall that only samples for which we have obtained a trace in both static and dynamic analysis, as reported in Table 7.2, are merged and evaluated. In family mode, the hybrid system, using traces produced by Monkey, achieves an F-measure of 0.88, whereas when using traces produced by humans, it achieves 0.87. When operating in package mode and using Monkey, it achieves an F-measure of 0.92, and 0.90 with humans, as reported in Table 7.3.

Note that we do not report code coverage in hybrid analysis because the traces from static analysis are an overestimation of the API calls in the app. Hence, merged traces do



(a) (b)

Figure 7.4: CDF of the percentage of code covered (a) when apps are stimulated by humans and Monkey, and (b) when the correctly classified and misclassified apps are stimulated by humans and Monkey.

not reflect code covered in each app when executed.

7.5 Comparative Analysis of Detection Approaches

We now set out to examine and compare: (1) the detection performance of each analysis method, i.e., detecting malware based on a behavioral model built via static, dynamic, or hybrid analysis, (2) the samples that are misclassified with each method, and (3) the samples misclassified in one method but correctly classified by another. Due to each method having inherent limitations, it is not clear from prior work how they compare against each other. Therefore, in this section, we shed light on their comparisons.

7.5.1 Detection Performance

We start by comparing the results of the different analysis methods. Recall that we have abstracted each API call to either its family or package name, therefore, each method operates in one of two modes. When operating in family mode, with static analysis we achieve an F-measure of 0.86, whereas with dynamic analysis, we achieve F-measure of 0.86 when apps are stimulated by Monkey and 0.85 when stimulated by humans (see Table 7.3).

In package mode, we achieve F-measure of 0.91 with static analysis, whereas with dynamic analysis we achieve F-measure of 0.92 when apps are stimulated by Monkey and 0.88 when stimulated by humans (see Table 7.3).

The results show that static analysis is at least as effective as dynamic analysis depending on the app stimulator used during dynamic analysis. We believe this is because the behavioral model used to perform detection primarily leverages API calls. Although static analysis is not able to detect maliciousness when code is loaded dynamically, it provides an overestimation of the API call sequences in the apk. Consequently, all behaviors that can be extracted from the apk are actually captured by the static analysis classifier. On the other hand, dynamic analysis captures only the behavior exhibited by the samples during runtime. Hence, any behavior not observed during runtime is not used in the decision-making of the dynamic analysis classifier.

To verify this hypothesis, we evaluate how the percentage of code covered differs when different app stimulators are employed as well as in correctly classified and misclassified samples. From Figure 7.4(a), we observe that when Monkey is used as the app stimulator, at least 48% of the API calls are triggered in 90% of the samples, compared to 35% when they are stimulated by humans. Similarly, as shown in Figure 7.4(b), 49% of the API calls are triggered in 90% of the samples correctly classified when Monkey is used to stimulate apps compared to 44% of API calls in 90% of the apps that are misclassified. When humans are used to stimulate the apps, 40% of the API calls are triggered in 90% of samples that are correctly classified compared to 38% triggered in 90% of samples misclassified. As a result of better code coverage, dynamic analysis performs better when apps are stimulated by Monkey compared to when apps are stimulated by crowdsourced users. Therefore, we find that, other than the non-susceptibility to evasion techniques such as dynamic code loading, dynamic analysis tools based on API calls may have no advantage over static analysis based tools unless the code coverage is improved.

However, when traces from static and dynamic analysis are merged into a hybrid sys-

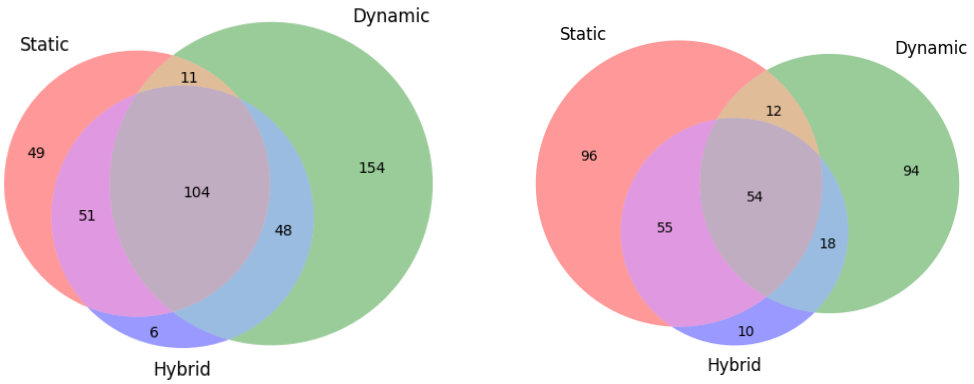
tem, in family mode, we achieve F-measure of 0.88 when the dynamic traces are produced by Monkey compared to 0.86 achieved by both static analysis and dynamic analysis (with Monkey) alone. Similarly, we achieve F-measure of 0.87 when the dynamic traces are generated with humans stimulating the apps compared to 0.86 and 0.85 achieved respectively by static and dynamic (humans) analysis alone. In package mode, the hybrid system achieves F-measure of 0.92 when the dynamic traces are produced by Monkey and 0.90 with humans. The hybrid system outperforms the dynamic analysis system in all modes (i.e., family and package), as it also captures behavior not exhibited during runtime execution of the apps as a result of the overestimation from static analysis, while it improves static analysis as it captures frequently used API calls – a behavior that cannot be captured by static analysis – and API calls that are dynamically loaded.

7.5.2 Misclassifications

Next, we examine the samples that are misclassified in each method of analysis, aiming to understand the differences in the model of the correctly classified and misclassified samples. We perform our analysis on samples that have been classified by all three methods in package mode.

7.5.2.1 Misclassifications Within Each Analysis Method

To understand why samples are misclassified, we formulate and verify the hypothesis that misclassifications are due to missing API calls that are considered "important" by the classifiers. To this end, we select the 100 most important features used by each classifier to distinguish between potential malware and benign samples, and evaluate the average number of these features present in each sample. We select the 100 most important features because it represents, at most, about 10% of the features recorded in our experiments. Recall that a feature in our detection technique is the probability of evoking an abstracted API call, and transitions not evoked during the experiments have probability of 0. The



(a) Humans (b) Monkey

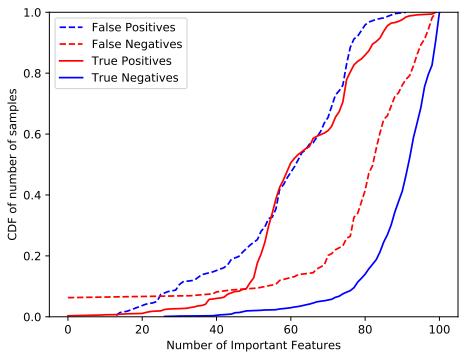
Figure 7.5: Number of false positives in each analysis method and when the apps are stimulated by humans (a) or Monkey (b) during dynamic analysis.

maximum number of features with probability > 0 in our dataset is 1,869 (static analysis) and the minimum is 1,022 (dynamic analysis with humans). We expect that samples that are misclassified will have a similar number of important features as those of the opposite class.

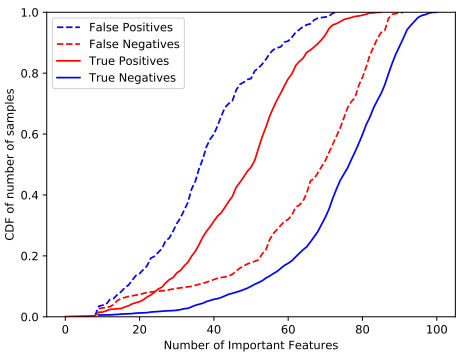
Using the top 100 features for each classifier, we compare the average number of the features in the true positives (i.e., correctly classified malware samples) to the false negatives (malware classified as benign), as well as true negatives to false positives.

False Positives.

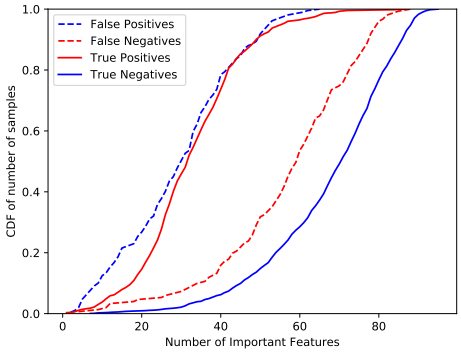
In Figure 7.5(a) and 7.5(b), we report the number of false positives in each method of analysis, respectively, when apps are stimulated by humans and by Monkey during dynamic analysis. With the former, there are 215, 317, and 209 false positives, respectively, with static, dynamic, and hybrid analysis. With the latter, there are 217, 178, and 137 false positives with static, dynamic, and hybrid analysis. Using the top 100 features, we find that the false positives in static analysis exhibit similar behavior to that observed in true positives. Specifically, they have, on average, 54.12 ± 22.65 features out of the 100 most important features, which is similar to 59.96 ± 19.46 in true positive samples. The same behavior is also observed in both dynamic and hybrid analysis irrespective of the app stimulator. In Figure 7.6, we plot the CDF of the number of features present in each



(a) Static Analysis



(b) Dynamic Analysis

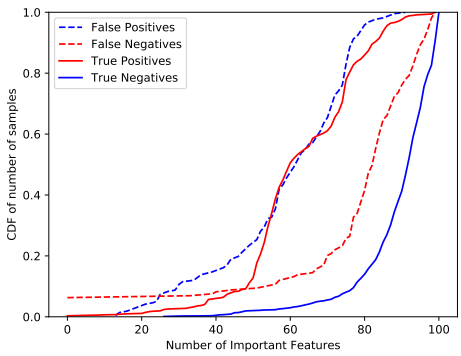


(c) Hybrid Analysis

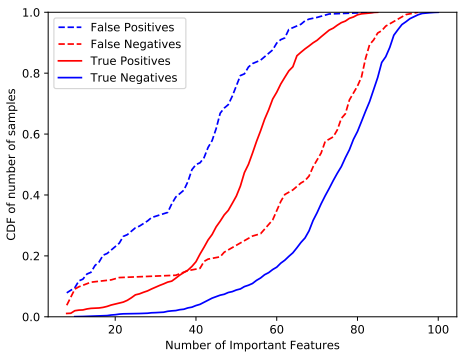
Figure 7.6: CDF of the number of features present (out of the 100 most important features) in each classification type for all analysis methods (with human during dynamic analysis).

classification type for all analysis methods when humans stimulate apps during dynamic analysis and, in Figure 7.7, with Monkey. The figures show that the behavioral model of the false positives in all analysis methods is similar to that observed on the true positives. For example, in Figure 7.6(c) (hybrid analysis) 90% of the false positives have no more than 50 of the 100 most important features (similar to the true positives – 49/100) while true negatives reach 86 features out of 100.

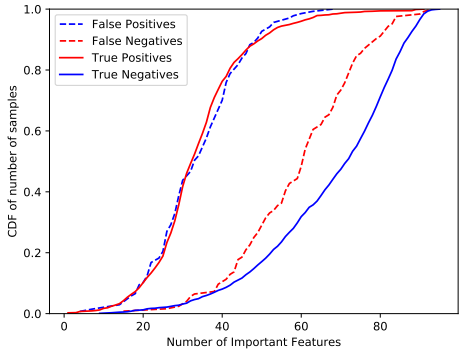
False Negatives. In Figure 7.8(a) and 7.8(b), we report the number of false negatives in each analysis method, respectively, when apps are stimulated by humans and Monkey. With the former, there are 148, 151, and 153 false negatives, respectively, in static, dynamic,



(a) Static Analysis



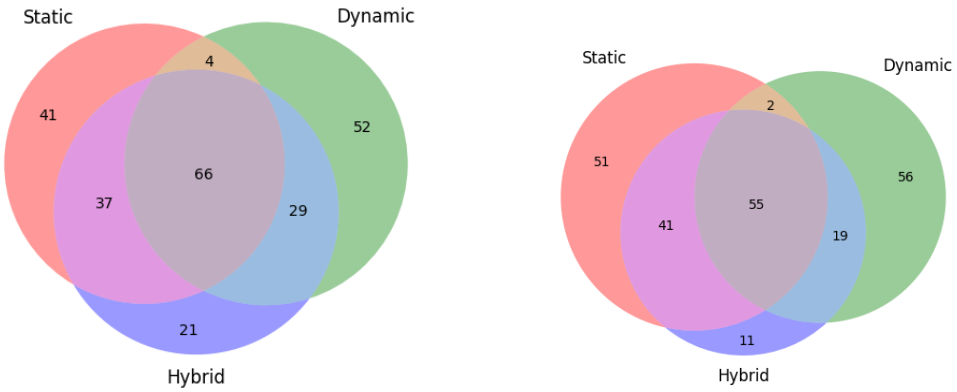
(b) Dynamic Analysis



(c) Hybrid Analysis

Figure 7.7: CDF of the number of features present (out of the 100 most important features) in each classification type for all analysis methods (with Monkey during dynamic analysis).

and hybrid analysis, while, with the latter, there are 149, 132, and 126 false negatives. In static analysis, we find that the behavioral model of the false negatives are similar to that observed in the true negatives. In particular, of the 100 most important features used to distinguish malware from benign samples, there are, on average, 82.08±11.75 features per false negative sample. The value is more similar to the 88.91±11.31 important features per true negative sample rather than the 59.96±19.46 important features per true positive sample. The same result is also observed in dynamic analysis irrespective of the stimulator, and in hybrid analysis as well. For example, in Figure 7.7(b) (dynamic analysis), 90% of the false negative samples have 84 of the 100 features, a value more similar to 89 features



(a) Humans (b) Monkey

Figure 7.8: Number of false negatives in each analysis method and when the apps are stimulated by humans (a) or Monkey (b) during dynamic analysis.

(true negatives) rather than 70 features (true positives).

7.5.2.2 Misclassifications Across Analysis Methods

Next, we attempt to clarify why some samples are misclassified by one method of analysis but correctly classified by another. The first important difference among the methods is the code coverage: dynamic analysis does not cover the entire code base of an app. Moreover, stimulating with Monkey vs humans yield different code coverage. This might result in a few scenarios:

1. Dynamic analysis may not have triggered the malicious code that is captured in static analysis;

2. Static analysis may reveal sequences of API calls that are not necessarily malicious, but characterize many malicious apps;

3. The API calls not triggered during dynamic analysis may poison the resulting Markov chains leading to training poisoning or misclassification of the sample, depending on whether the sample is part of the training or the test set.

Scenarios (1) and (2) are possible reasons why static analysis correctly detects some

samples and dynamic analysis does not, while (3) refers to the opposite. Although the hybrid system captures sequences of API calls from both static and dynamic analysis, it actually results in completely new Markov chains and features for training and classification. While more accurate than the individual methods, as the features values change (i.e., the transition probabilities), it behaves differently.

Another important factor is the presence of loops in the code waiting for user interaction. A good example in our dataset are the games *Dumb ways to die 1 & 2*. These apps have "minigames" where a user has to click several times on the right spot of the screen at the right time. When executed, the apps enter a loop waiting for user action, and decide the next action based on what happened before returning to waiting for user action. Static analysis would catch the four different outcomes (i.e., execution path) of the loop, i.e., wrong click, correct click, the user won the game, the user lost the game. Dynamic analysis would repeat the loop many times depending on the continuous clicks of the human or of Monkey, and the user/Monkey may never win or lose. Static analysis will record the four possible loop paths without repeating the sequences in its traces, and the user/Monkey may not record all the possible sequences, but have duplicated sequences due to multiple clicks resulting in the same outcome. All these differences characterize the recorded traces, and therefore may result in different Markov chains and decisions among the methods.

7.6 Discussion

In this chapter, we address three research questions: 1) can we extend static analysis-based malware detection techniques based on behavioral modeling to use dynamic analysis, 2) does having humans provide input during dynamic analysis improve malware detection compared to pseudorandom input generators, and 3) how does each analysis method compare to the other when using a common modeling approach. To this end, we built a dynamic analysis tool, AuntieDroid, which supports app stimulation via both humans (via crowd-

sourcing) and pseudorandom input generators (Monkey). We also reuse MaMaDroid with the most recent packages from API level 26. Then, to build a hybrid system, we merged the sequences of API calls from static and dynamic analysis. All three methods operate in one of two modes, i.e., family and package, based on the level of abstraction; in family mode, static, dynamic (human/Monkey), and hybrid analysis, respectively, achieve F-measures of 0.86, 0.85/0.86, and 0.88. Whereas in package mode, they achieve 0.91, 0.88/0.92, and 0.92.

Overall, our experiments yield some interesting findings. Hybrid analysis performs best because it captures the best of both worlds, i.e., static and dynamic analysis, as it is able to capture the sequences of API calls that are actually executed and/or dynamically loaded (from the latter), and capture code not executed during testing due to code overestimation (from the former). Nonetheless, static analysis performs well overall, often better than dynamic analysis; when looking at misclassifications across methods, we find that those occurring in dynamic but not in static analysis are likely due to poor code coverage, thus, the feature vectors in dynamic analysis may not reveal features (e.g., a chunk of benign code in repackaged samples) that characterize malware in our dataset. Then, we show that dynamic analysis performs better with Monkey than humans because the former is able to trigger more code than the latter.

Cost Analysis Between App Stimulators. When humans are recruited to test apps, we pay each user $0.12 per session (a session involves a user testing a set of four randomly selected apps). For the 5,030 users we recruited, we paid a total of $603.6. Whereas when Monkey is used to test apps, we do not incur any cost other than the memory and CPU requirement for running each Monkey instance. With respect to the average number of API calls each stimulator trigger per seconds, as stated in Section 7.2.2, we set Monkey to trigger an event at a rate of 100ms (i.e., 10 events/s) for 5 minutes. As an event can map to multiple API calls, we find that Monkey triggers on average, about 99 API calls per

second.[4] With humans, we do not restrict how long each users should test the apps and we did not keep track of how long each user spends on an app before moving on to the next app. Hence, it is not possible to calculate the number of API calls triggered per second by our users. Nonetheless, we show in Figure 7.4(a), the CDF of the percentage of the API calls in app users are able to trigger compared to Monkey.

7.6.1 Challenges with Dynamic Analysis

We now discuss the challenges we faced when analyzing the apps during dynamic analysis.

Apps with no Trace. When executing apps during dynamic analysis on AuntieDroid, we find that, for some apps, we do not obtain any trace. Specifically, there are no traces for 724 apps when using Monkey and 693 when using humans. In total, we do not obtain traces for 835 unique apps. Further examination of the logs of these apps shows that this happens because:

1. The apps stop responding, i.e., "Application Not Responding" (ANR) is thrown by Android as the app cannot respond to user input (154 occurrences);
2. Lack of OpenGL minimum specification matching the requirement of the app (229);
3. Fatal error in an app's `onCreate()` launcher activity method, causing the app to crash (452).

Apps with no Package Name in Apk. For some apps where we do obtain traces, we find that they do not contain any API call beginning with the package name of the app. Recall that in order to evaluate the code coverage of each app, we focus on API calls that begin with the package name of the app. Using this approach, we find there are 350 apps in our dataset for which calls beginning with the package name are not present in the dynamic traces nor the apk. Upon further analysis, we find these apps use package names that are

[4]Note that this rate takes into account apps that crash or become non-responsive, hence, it is dependent on several factors such as failures, type of events etc.

different from that declared in their manifest. To calculate code coverage for these apps, we use as package names, those packages (excluding packages from the Android and Google API) that have at least one activity and broadcast receiver and/or service classes in the apk.

7.6.2 Limitations

One of the contributions of our work is extending CHIMP [17] to create a virtual device on which AUNTIEDROID builds, but, naturally, our work is not without limitations. First, we only support one Android version at this time: KitKat 4.4.2. However, we expect KitKat to actually be more vulnerable to malware than more recent versions of Android; moreover, only 0.8% of apps in our sample (see Section 7.3.3) require a newer version of Android. An interesting future research avenue would be to explore differences in malware behavior across versions of Android, which previous work has indicated to be quite significant [216].

Second, our virtual device does not yet support the full range of OpenGL operations, which somewhat limits the number of apps that we can evaluate. However, this support is an ongoing effort by the Android x86 community[5] and only 229 (4.6% of our dataset) of apps that we failed to acquire a trace for required full OpenGL support.

Third, when performing static analysis with MAMADROID, we do not analyze all apps because some apps fail to decompile. These failures are due to: bad CRC-32 redundancy checks, errors during unpacking, or failure to transform the Java bytecode into jimple intermediate representation by Soot. We exclude these apps from our analysis and report static analysis results for only apps that we have successfully decompiled. Finally, we remark that, when stimulating the apps using humans, the users we recruit may not have tested each app in the way they would have used the apps on their own device, which is also an item for future work.

Although some characteristics peculiar to AUNTIEDROID's virtual device (e.g., it runs

[5]Qemu's OpenGL 3 support was announced very recently – see `http://www.android-x86.org/releases/releasenote-7-1-rc1`

as a hardware assisted virtualization) should prevent evasion by malware that tries to circumvent emulators/virtual devices using environment variables [70, 124, 142, 164], we plan, as part of future work, to update it to use a virtual device that appears as close to a real device as possible. Moreover, we intend to use input generators that target specific behaviors of an app, so as to target certain API calls mostly used by malware rather than trying to improve the code coverage during dynamic analysis. Finally, we plan to detect and measure the prevalence of malware that specifically employs dynamic code loading as an evasion technique.

7.6.3 Concluding Remarks

In this chapter, we presented AuntieDroid for detecting malware dynamically by integrating MaMaDroid into Chimp. We allowed both human users and automated pseudo-random input generator i.e., Monkey to instantiate AuntieDroid. We found that Monkey outperforms human users largely due to its ability to achieve better code coverage. We also performed a comparative analysis of the different analytical methods i.e., static, dynamic, and hybrid in malware detection, showing that hybrid analysis performs best and that static analysis is at least as effective as dynamic analysis depending on the input generator to apps during dynamic analysis. Our results encourage the use of hybrid analysis in the detection of malware as it inherits the pros of both static and dynamic analysis, while addressing some of the cons inherent in one of static or dynamic analysis that the other addresses.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

With the increasing number of Android apps and sources of personally identifiable information on the devices these apps run on, it is imperative to analyze how these apps access and transmit these information. Prior work showed that, due to vulnerabilities in these apps or the Android OS itself, malevolent actors can exfiltrate users' information thereby, compromising users' security and privacy. Similarly, other research efforts showed that some apps compromise users' security and privacy because they are designed to be malicious. This thesis builds on prior work by measuring and proposing new mitigation techniques to the security and privacy issues in Android apps.

In particular, we first examined whether popular apps still use vulnerable SSL sockets that allow man-in-the-middle (MiTM) attackers retrieve user information during transmission (Chapter 4). Our analysis showed that this practice is still common even after more than two years since studies highlighted the security and privacy risks of such practice. The vulnerable apps allow MiTM attacks because they either accepted all self-signed certificates or did not verify the domain name. In order to mitigate MiTM attacks, we recommended (and provided code sample) that app developers use a certificate pinning logic that verifies both recognized CA-signed and self-signed certificates correctly and allows them employ self-signed certificates safely.

Next, we investigated whether apps that are advertised as employing end-to-end encryption, anonymity, or ephemerality, actually do so and do so correctly (Chapter 5). We

showed that some "anonymous" social networking apps (Whisper and Yik Yak) actually identify their users with distinct IDs that are persistent as previous activities like chats, whispers and yaks are restored to the device even if the user uninstalls and reinstalls the app. This behavior shows that, although they do not require users to provide their email or phone number, they can still persistently link – and possibly de-anonymize – users. Similarly, ephemeral messaging apps may not always delete messages as we found in Snapchat. While Snapchat promises that messages will "disappear" after 10 seconds, the messages are not immediately deleted from their servers, as old messages are actually included in responses sent to the clients. However, we confirmed that apps such as Telegram and Wickr do offer end-to-end encrypted chat messaging.

Then, we presented MaMaDroid, an Android malware detection system that is based on modeling the sequences of API calls as Markov chains (Chapter 6). MaMaDroid is designed to operate in one of three modes, with different granularities, by abstracting API calls to either families, packages or classes. It effectively detects unknown malware samples developed around the same time as the samples on which it is trained (F-measure up to 0.99), and maintains a high detection performance over time at F-measure of 0.86 and 0.75, respectively, one and two years after the model has been trained. We examined how the modeling approach effects the performance of MaMaDroid by comparing it to a variant, FAM, that employs frequency analysis modeling instead. Our results show that Markov chain-based modeling achieves higher detection accuracy and is more robust, especially in scenarios where API calls are not more frequent in malware than in benign apps. Similarly, we also examined the effects of abstraction by comparing FAM to DroidApiMiner [14], since both are based on frequency analysis. FAM outperforms DroidApiMiner showing that API call abstraction improves the detection accuracy of our systems. Overall, our results demonstrate that statistical behavioral models introduced by MaMaDroid– in particular, abstraction and Markov chain modeling of API call sequence – are more robust than traditional techniques, highlighting how it can form the basis of more advanced

detection systems in the future.

Finally, we compared how each program analysis approach (i.e., static, dynamic, and hybrid) performs when they employ the same underlying modeling technique (Chapter 7). Specifically, we employ MaMaDroid as the underlying modeling technique and implement it in a dynamic setting by integrating it in a virtual device. The resulting tool is called AuntieDroid and it supports app stimulation via both humans and pseudorandom input generators (Monkey). To build a hybrid system, we merged the sequences of API calls from static and dynamic analysis. Overall, our experiments showed that hybrid analysis performs best because it captures the best of static and dynamic analysis, as it is able to capture the sequences of API calls that are actually executed and/or dynamically loaded (from the latter), and capture codes that are not executed during testing due to code overestimation (from the former). Nonetheless, static analysis performs well overall, often better than dynamic analysis; when looking at misclassifications across methods, we found that those occurring in dynamic but not in static analysis are likely due to poor code coverage, thus, the feature vectors in dynamic analysis may not reveal features (e.g., a chunk of benign code in repackaged samples) that characterize malware in our dataset. We also showed that dynamic analysis performs better with Monkey than humans because the former is able to trigger more code than the latter.

In order to help developers mitigate against the problems highlighted in this thesis, we make our code publicly available. Developers or even app stores that want to detect vulnerable TLS implementation in Android apps can do so using our fork of Mallodroid [82], available here: `https://github.com/luckenzo/mallodroid`. As Mallodroid does not correctly analyze multidex apps and cannot tell when an app has a secure implementation, we have developed a tool called TLSDroid, which performs a large scale analysis of apps as well as address the shortcomings of MalloDroid. TLSDroid will also be made available on the author's profile[1] on Github. In addition, we also make our sample code for safe use

[1] `https://github.com/luckenzo`

of self-signed certificates as a Github gist[2], so it is easily accessible when developers search for selected keywords.

With respect to detecting malicious apps, we have also made the code of MaMaDroid as well as hashes of our dataset available in the project's online repository: `https://bitbucket.org/gianluca_students/mamadroid_code`. As Chimp is offered as a paid service[3] by its developers, we do not make AuntieDroid available. Nonetheless, the modifications we make to Chimp that allow us collect method traces are preserved and are accessible in the paid service. Hence, researchers can use Chimp to collect the traces and use the MaMaDroid scripts to parse the traces and perform classification. Finally, researchers can also modify how MaMaDroid reports performance (e.g., from F-measure to Precision) to fit their use-case. For example, in scenarios where the prediction of a class (e.g., the malware class) is of utmost importance, Precision can be used and the classification model can be tuned to improve Precision and reduce false negative rate.

8.2 Future Work

In order to perform MiTM attacks as presented in Chapter 4 and 5, we install a CA certificate on the device which is then used as one of the trust anchors (i.e., trusted certificates) by the Android OS to determine whether a certificate chain should be accepted as valid or not. But as from API level 24, this behavior has changed.[4] Any app targeting this API level or above, will no longer trust CA certificates installed by users or apps with administrative permissions by default, except the app explicitly decides to. One possible work that arises from this change is to examine if this change has been implemented correctly without vulnerabilities, and most importantly, if the change has been implemented correctly, do developers of apps that handle financial information (or apps advertised as

[2] `https://gist.github.com/luckenzo/416d9da141b302bfcfcd7b0d3b95489f`
[3] https://tappas.io/
[4] `https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html`

security-enhancing) trust user installed CAs without the use of certificate pinning?

Future work could also investigate how piggybacking (i.e., injecting malicious code into benign apps and repackaging and distributing it as the original) affects the effectiveness of MaMaDroid and other malware detection tools. For example, if a benign app is repackaged and only a single method that performs malicious activity is injected, does MaMaDroid detect the app as malicious? If more methods and classes that perform malicious activities are added, is there a threshold of the number of methods or classes at which MaMaDroid detects an app as malicious and benign otherwise?

Finally, future work could address the limitations present in this thesis. For example, they could use other decompilers or custom tools (instead of Soot) to extract call graphs from apps, so as to ensure that all apps are analyzed.

Bibliography

[1] Android Advertising ID. https://developer.android.com/google/play-services/id.html.

[2] AndroidRank. http://www.androidrank.org/.

[3] HostnameVerifier. https://developer.android.com/reference/javax/net/ssl/HostnameVerifier.html.

[4] HttpsURLConnection. https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html.

[5] Product Hunt – Anonymous Apps. http://www.producthunt.com/e/anonymous-apps.

[6] SSLSocketFactory. https://developer.android.com/reference/javax/net/ssl/SSLSocketFactory.html.

[7] SSLsplit – Transparent and Scalable SSL/TLS Interception. http://www.roe.ch/SSLsplit.

[8] Telegram: "If you don't trust us, use the secret chats". http://features.en.softonic.com/telegram-secret-chats.

[9] X509TrustManager. https://developer.android.com/reference/javax/net/ssl/X509TrustManager.html.

[10] Global Internet User Survey Summary Report. http://www.internetsociety.org/sites/default/files/rep-GIUS2012global-201211-en.pdf, 2012.

[11] Android Security Bulletin. https://source.android.com/security/bulletin/, Accessed August 15, 2018.

[12] CVE Details (The Ultimate Security Vulnerability Datasource. Android: Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html, Accessed August 15, 2018.

[13] Smartphone Users Worldwide Will Total 1.75 Billion in 2014. http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536, Accessed January, 2015.

[14] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm)*, 2013.

[15] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking For: The Impact of Information Sources on Code Security. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[16] D. Akhawe and A. P. Felt. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

[17] M. Almeida, M. Bilal, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Varvello, and J. Blackburn. CHIMP: Crowdsourcing Human Inputs for Mobile Phones. In *Proceedings of The Web Conference*, 2018.

[18] A. Amamra, C. Talhi, and J.-M. Robert. Smartphone Malware Detection: From a Survey Towards Taxonomy. In *Proceedings of the 7th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2012.

[19] C. Amrutkar, P. Traynor, and P. C. van Oorschot. Measuring SSL Indicators on Mobile Browsers: Extended Life, or End of the Road? In *Proceedings of the International Conference on Information Security (ISC)*, 2012.

[20] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[21] P. Andriotis, M. A. Sasse, and G. Stringhini. Permissions Snapshots: Assessing Users' Adaptation to the Android Runtime Permission Model. In *Proceedings of the IEEE Workshop on Information Forensics and Security (WIFS)*, 2016.

[22] AndroidRank. Android Market App Statistics. https://goo.gl/UvOo0M, Accessed April 12, 2017.

[23] Apktool. A Tool for Reverse Engineering Android Apk Files. `https://ibotpeaches.github.io/Apktool/`, 2017.

[24] AppBrain. Number of Android Applications. `https://www.appbrain.com/stats/number-of-android-apps`, August 2018.

[25] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[27] K. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[28] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[29] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. Butler, and A. Alkhelaifi. Securing SSL Certificate Verification Through Dynamic Linking. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[30] H. Berghel and J. Uecker. WiFi Attack Vectors. *Communications of the ACM*, 48(8):21–28, 2005.

[31] S. Bernard, S. Adam, and L. Heutte. Using Random Forests for Handwritten Digit Recognition. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition (ICDAR)*, 2007.

[32] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[33] N. Bilton. Why I Use Snapchat: It's Fast, Ugly and Ephemeral. http://nyti.ms/1jBMZrQ, 2014.

[34] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[35] D. Bradbury. Hacking WiFi the Easy Way. *Network Security*, 2011(2):9–12, 2011.

[36] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[37] J. Brownlee. Support Vector Machines for Machine Learning. https://machinelearningmastery.com/support-vector-machines-for-machine-learning/, 2016.

[38] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[39] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2012.

[40] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[41] P. Calciati and A. Gorla. How Do Apps Evolve in Their Permission Requests?: A Preliminary Study. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE Press, 2017.

[42] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detecting Android Malware Using Sequences of System Calls. In *Proceedings of the Workshop on Software Development Lifecycle for Mobile*, 2015.

[43] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Acquiring and Analyzing App Metrics for Effective Mobile Malware Detection. In *Proceedings of the ACM International Workshop on Security and Privacy Analytics (IWSPA)*, 2016.

[44] G. Canfora, F. Mercaldo, and C. A. Visaggio. An HMM and Structural Entropy Based Detector for Android Malware: An Empirical Study. *Computers & Security*, 61:1–18, 2016.

[45] N. Cardozo, G. Gebhart, and E. Portnoy. Secure Messaging? More Like A Secure Mess. https://www.eff.org/deeplinks/2018/03/secure-messaging-more-secure-mess, 2018.

[46] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2016.

[47] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-Scale Mobile Malware Analysis. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.

[48] Check Point. ExpensiveWall: A Dangerous 'Packed' Malware On Google Play that will Hit Your Wallet. https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/, 2017.

[49] Check Point. FalseGuide misleads users on GooglePlay. https://blog.checkpoint.com/2017/04/24/falaseguide-misleads-users-googleplay/, 2017.

[50] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th ACM International Conference on Software Engineering (ICSE)*, 2014.

[51] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[52] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang. AR-Miner: Mining Informative Reviews for Developers from Mobile App Marketplace. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014.

[53] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li. Automated Poisoning Attacks and Defenses in Malware Detection Systems: An Adversarial Machine Learning Approach. *Computers & Security*, 73:326–344, 2018.

[54] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware. In *Proceedings of the ACM ASIA Conference on Information, Computer and Communications Security (AsiaCCS)*, 2016.

[55] W. Chen, D. Aspinall, A. D. Gordon, C. Sutton, and I. Muttik. More Semantics More Robust: Improving Android Malware Classifiers. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2016.

[56] Y. Chen, M. Ghorbanzadeh, K. Ma, C. Clancy, and R. McGwier. A Hidden Markov Model Detection of Malicious Android Applications at Runtime. In *Proceedings of the Wireless and Optical Communication Conference (WOCC)*, 2014.

[57] N. Cheng, X. O. Wang, W. Cheng, P. Mohapatra, and A. Seneviratne. Characterizing Privacy Leakage of Public WiFi Networks for Users on Travel. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013.

[58] S. R. Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015.

[59] J. Clay. Continued Rise in Mobile Threats for 2016. `http://blog.trendmicro.com/continued-rise-in-mobile-threats-for-2016/`, 2016.

[60] Confide Inc. Your Off-The-Record Messenger. `https://getconfide.com/`, 2015.

[61] M. Conti, N. Dragoni, and S. Gottardo. Mithys: Mind the Hand You Shake – Protecting Mobile Devices from SSL Usage Vulnerabilities. In *Proceedings of the International Workshop on Security and Trust Management (STM)*, 2013.

[62] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. `http://www.ietf.org/rfc/rfc5280.txt`, 2008.

[63] P. Cousot. Abstract Interpretation Based Formal Methods and Future Challenges, (Invited Paper). In *Informatics – 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[64] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2012.

[65] E. Cunningham. Keeping You Safe with Google Play Protect. https://www.blog.google/products/android/google-play-protect/, 2017.

[66] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. NetworkProfiler: Towards Automatic Fingerprinting of Android Apps. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013.

[67] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The Tangled Web of Password Reuse. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[68] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the International Conference on Information Security (ISC)*, 2011.

[69] Ddmlib: APIs for Talking with Dalvik VM. https://mvnrepository.com/artifact/com.android.ddmlib/ddmlib, 2017.

[70] W. Diao, X. Liu, Z. Li, and K. Zhang. Evading Android Runtime Analysis Through Detecting Programmed Interactions. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2016.

[71] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, volume 31, 2011.

[72] J. Drake. Stagefright: Scary Code in the Heart of Android. *BlackHat USA*, 2015.

[73] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[74] A. Egners, U. Meyer, and B. Marschollek. Messing with Android's Permission Model. In *Proceedings of the IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom)*, 2012.

[75] EMMA. EMMA: A Free Java Code Coverage Tool. http://emma.sourceforge.net/, 2017.

[76] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[77] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[78] Ericsson. Ericsson Mobility Report. https://bit.ly/2LueXHm, June 2015.

[79] Ericsson. Ericsson Mobility Report. https://goo.gl/FhUhan, June 2016.

[80] N. S. Evans, A. Benameur, and Y. Shen. All Your Root Checks Are Belong to Us: The Sad State of Root Detection. In *Proceedings of the 13th ACM International Symposium on Mobility Management and Wireless Access*, 2015.

[81] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith. Hey, NSA: Stay Away from My Market! Future Proofing App Markets Against Powerful Attackers. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[82] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[83] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking SSL Development in an Appified World. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[84] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2015.

[85] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[86] A. P. Felt et al. Improving SSL Warnings: Comprehension and Adherence. In *Proceedings of the ACM A Conference on Human Factors in Computing Systems*, 2015.

[87] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, 2011.

[88] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand. Automated Synthesis of Semantic Malware Signatures Using Maximum Satisfiability. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.

[89] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Android UI Deception Revisited: Attacks and Defenses. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 2016.

[90] A. Ferreira, J.-L. Huynen, V. Koenig, G. Lenzini, and S. Rivas. Socio-Technical Study on the Effect of Trust and Context when Choosing WiFi Names. In *Proceedings of the International Workshop on Security and Trust Management*. Springer, 2013.

[91] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[92] E. Fix and J. Hodges. Discriminatory Analysis, Non-Parametric Discrimination. *USAF School of Aviation Medicine*, 31, 1951.

[93] J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics, 2nd edition, 2008.

[94] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh. Why People Hate Your App: Making Sense of User Feedback in a Mobile App Store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013.

[95] F. Gagnon, M.-A. Ferland, M.-A. Fortier, S. Desloges, J. Ouellet, and C. Boileau. AndroSSL: A Platform to Test Android Applications Connection Security. In *Proceedings of the International Symposium on Foundations and Practice of Security*. Springer, 2015.

[96] S. Gallagher. Intercepted WhatsApp Messages Led to Belgian Terror Arrests. `http://arstechnica.co.uk/tech-policy/2015/06/intercepted-whatsapp-messages-led-to-belgian-terror-arrests/`, 2015.

[97] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek. Obfuscation-Resilient, Efficient, and Accurate Detection and Family Identification of Android Malware. Technical report, 2015.

[98] Gartner, Inc. Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013. `http://www.gartner.com/newsroom/id/2665715`, 2015.

[99] Gartner, Inc. Worldwide Tablet Sales Grew 68% in 2013, with Android Capturing 62% of the Market. http://www.gartner.com/newsroom/id/2674215, 2015.

[100] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security (AISec)*, 2013.

[101] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[102] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *ACM Sigplan Notices*, 40(6):213–223, 2005.

[103] S. Gold. Cracking Wireless Networks. *Network Security*, 2011(11):14–18, 2011.

[104] M. Gomez, M. Martinez, M. Monperrus, and R. Rouvoy. When App Stores Listen to the Crowd to Fight Bugs in the Wild. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, volume 2, 2015.

[105] Google. Android Security 2017 Year in Review. `https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf`, year=2018.

[106] Google. Google Play Developer Program Policies. `https://play.google.com/about/developer-content-policy.html`, 2015.

[107] Google. UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey.html`, 2017.

[108] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[109] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-Day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[110] N. Hardy. The Confused Deputy:(or Why Capabilities Might Have Been Invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[111] J. Hare, L. Hartung, and S. Banerjee. Beyond Deployments and Testbeds: Experiences with Public Usage on Vehicular WiFi Hotspots. In *Proceedings of the 10th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[112] R. Hay and A. Dayan. Android Keystore Stack Buffer Overflow, 2014.

[113] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf. Support Vector Machines. *IEEE Intelligent Systems and their Applications*, 13(4):18–28, 1998.

[114] S. Heuser, M. Negro, P. K. Pendyala, and A.-R. Sadeghi. DroidAuditor: Forensic Analysis of Application-Layer Privilege Escalation Attacks on Android (Short Paper). In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 2016.

[115] T.-H. Ho, D. Dean, X. Gu, and W. Enck. PREC: Practical Root Exploit Containment for Android Devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2014.

[116] J. Hodges and P. Saint-Andre. Representation and Verification of Domain-Based Application Service Identity Within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). `http://tools.ietf.org/html/rfc6125`, 2015.

[117] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu. HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[118] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu. Performance Evaluation on Permission-Based Detection for Android Malware. In *Proceedings of the Advances in Intelligent Systems and Applications*, 2013.

[119] C. Iacob and R. Harrison. Retrieving and Analyzing Mobile Apps Feature Requests from Online Reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013.

[120] IDC. Smartphone OS Market Share, 2016 Q3. `http://www.idc.com/promo/smartphone-market-share/os`, November 2016.

[121] M. Ikram, N. Vallina-Rodriguez, S. Seneviratne, M. A. Kaafar, and V. Paxson. An Analysis of the Privacy and Security Risks of Android VPN Permission-Enabled Apps. In *Proceedings of the 2016 ACM Internet Measurement Conference (IMC)*, 2016.

[122] B. Ives, K. R. Walsh, and H. Schneider. The Domino Effect of Password Reuse. *Communications of the ACM*, 47(4):75–78, 2004.

[123] S. Jansen and E. Bloemendal. Defining App Stores: The Role of Curated Marketplaces in Software Ecosystems. In *Proceedings of the International Conference of Software Business*. Springer, 2013.

[124] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[125] I. Jolliffe. *Principal Component Analysis*. John Wiley & Sons, Ltd, 2002.

[126] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting Concept Drift in Malware Classification Models. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[127] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What Do Mobile App Users Complain About? *IEEE Software*, 32(3):70–77, 2015.

[128] Y. Kikuchi, H. Mori, H. Nakano, K. Yoshioka, T. Matsumoto, and M. van Eeten. Evaluating Malware Mitigation by Android Market Operators. In *9th Workshop on Cyber Security Experimentation and Test (CSET)*. USENIX Association, 2016.

[129] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security)*, 2009.

[130] M. Kranch and J. Bonneau. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[131] J. Kraunelis, X. Fu, W. Yu, and W. Zhao. A Framework for Detecting and Countering Android UI Attacks via Inspection of IPC Traffic. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2018.

[132] N. Lageman, M. Lindsey, and W. Glodek. Detecting Malicious Android Applications from Runtime Behavior. In *Proceedings of the IEEE Military Communications Conference (MILCOM)*, 2015.

[133] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. In *Proceedings of the 39th IEEE Annual Computer Software and Applications Conference (COMPSAC)*, 2015.

[134] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer. ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[135] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis. AndRadar: Fast Discovery of Android Applications in Alternative Markets. In *Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2014.

[136] H. Lockheimer. Android and Security. `https://googlemobile.blogspot.com/2012/02/android-and-security.html`, 2012.

[137] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2014.

[138] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Proceedings of the International Symposium on Foundations and Practice of Security*. Springer, 2012.

[139] W. Maalej and H. Nabil. Bug Report, Feature Request, or Simply Praise? On Automatically Classifying App Reviews. In *Proceedings of the 23rd International Requirements Engineering Conference (RE)*. IEEE, 2015.

[140] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.

[141] D. J. MacKay and D. J. Mac Kay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[142] D. Maier, T. Müller, and M. Protsenko. Divide-and-Conquer: Why Android Malware Cannot Be Stopped. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2014.

[143] J. Mangalindan. Yahoo, like Google, Plans Encrypted Email. http://for.tn/1ElU0SC, 2014.

[144] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.

[145] D. Marino and T. Millstein. A Generic Type-and-Effect System. In *Proceedings of the 4th ACM International Workshop on Types in Language Design and Implementation*, 2009.

[146] M. Miettinen, S. Heuser, W. Kronz, A.-R. Sadeghi, and N. Asokan. ConXsense: Automated Context Classification for Context-Aware Access Control. In *Proceedings of the ACM ASIA Conference on Information, Computer and Communications Security (AsiaCCS)*, 2014.

[147] C. Miller. Mobile Attacks and Defense. *IEEE Security & Privacy*, 9(4):68–70, 2011.

[148] O. Mirzaei, G. Suarez-Tangil, J. Tapiador, and J. M. de Fuentes. TriFlow: Triaging Android Applications Using Speculative Information Flows. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2017.

[149] D. Morris. An Extremely Convincing WhatsApp Fake Was Downloaded More than 1 Million Times from Google Play. http://fortune.com/2017/11/04/whatsapp-fake-google-play/, 2017.

[150] Y. Y. Ng, H. Zhou, Z. Ji, H. Luo, and Y. Dong. Which Android App Store Can Be Trusted in China? In *Proceedings of the 38th IEEE Annual Computer Software and Applications Conference (COMPSAC)*, 2014.

[151] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2015.

[152] M. Niemietz and J. Schwenk. UI Redressing Attacks on Android Devices. *BlackHat Abu Dhabi*, 2012.

[153] J. R. Norris. *Markov Chains*. Cambridge University Press, 1998.

[154] J. Oberheide and C. Miller. Dissecting the Android Bouncer. *SummerCon*, 2012.

[155] P. H. O'Neill. The State of Encryption Tools, 2 Years After Snowden Leaks. http://www.dailydot.com/politics/encryption-since-snowden-trending-up/, 2015.

[156] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro. A Family of Droids – Android Malware Detection via Behavioral Modeling: Static vs Dynamic Analysis. In *Proceedings of the 16th IEEE Annual Conference on Privacy, Security and Trust (PST)*, 2018.

[157] L. Onwuzurike and E. De Cristofaro. Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2015.

[158] L. Onwuzurike and E. De Cristofaro. Experimental Analysis of Popular Smartphone Apps Offering Anonymity, Ephemerality, and End-to-End Encryption. In *Proceedings of the NDSS Workshop on Understanding & Enhancing Online Privacy*, 2016.

[159] L. Onwuzurike and E. De Cristofaro. POSTER: Experimental Analysis of Popular Anonymous, Ephemeral, and End-to-End Encrypted Apps. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2016.

[160] L. Onwuzurike, E. Mariconti, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). *arXiv preprint arXiv:1711.07477*, 2017.

[161] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How Can I Improve My App? Classifying User Reviews for Software Maintenance and Evolution. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015.

[162] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[163] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. *arXiv preprint arXiv:1807.07838*, 2018.

[164] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security (EuroSec)*, 2014.

[165] A. Pfitzmann and M. Hansen. A Terminology for Talking About Privacy by Data Minimization. `https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf`, 2010.

[166] M. Pielot and N. Oliver. Snapchat: How to Understand a Teen Phenomenon. In *ACM CHI Workshop on Understanding Teen UX*, 2014.

[167] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[168] I. Polakis, M. Diamantaris, T. Petsas, F. Maggi, and S. Ioannidis. Powerslave: Analyzing the Energy Consumption of Mobile Antivirus Software. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2015.

[169] B. Potter. Wireless Hotspots: Petri Dish of Wireless Security. *Communications of the ACM*, 49(6):50–56, 2006.

[170] Profiling with Traceview and Dmtracedump. `https://developer.android.com/studio/profile/traceview.html`, 2017.

[171] S. Rasthofer, S. Arzt, and E. Bodden. A Machine-Learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[172] S. Rasthofer, S. Arzt, M. Kolhagen, B. Pfretzschner, S. Huber, E. Bodden, and P. Richter. Droidsearch: A Tool for Scaling Android App Triage to Real-world App Stores. In *Proceedings of the IEEE Science and Information Conference (SAI)*, 2015.

[173] V. Rastogi, Y. Chen, and X. Jiang. Catch Me if You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security (TIFS)*, 9(1):99–108, 2014.

[174] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS Usage in Android Apps. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 2017.

[175] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, et al. *droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Computing Surveys (CSUR)*, 49(3):55, 2016.

[176] C. Ren, P. Liu, and S. Zhu. Windowguard: Systematic Protection of GUI Security in Android. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.

[177] F. Roesner, B. T. Gill, and T. Kohno. Sex, Lies, or Kittens? Investigating the Use of Snapchat's Self-Destructing Messages. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. 2014.

[178] A. Sadeghi. *Efficient Permission-Aware Analysis of Android Apps*. PhD thesis, University of California, Irvine, 2017.

[179] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. `https://www.ietf.org/rfc/rfc6960.txt`, 2013.

[180] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez. PUMA: Permission Usage to Detect Malware in Android. In *Proceedings of the International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*, 2013.

[181] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. MADAM: Effective and Efficient Behavior-Based Android Malware Detection and Prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, 2018.

[182] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android Permissions: A Perspective Combining Risks and Benefits. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2012.

[183] F. Sebastiani. Machine Learning in Automated Text Categorization. *ACM Computing Surveys (CSUR)*, 34(1):1–47, 2002.

[184] J. Sellwood and J. Crampton. Sleeping Android: The Danger of Dormant Permissions. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 55–66. ACM, 2013.

[185] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti. Predicting User Traits from a Snapshot of Apps Installed on a Smartphone. *ACM SIGMOBILE Mobile Computing and Communications Review*, 18(2):1–8, 2014.

[186] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A Comprehensive Security Assessment. *IEEE Security & Privacy*, (2):35–44, 2010.

[187] E. Shein. Ephemeral Data. *Communications of the ACM*, 56(9):20–22, 2013.

[188] Snapchat. Privacy Policy. `https://www.snapchat.com/privacy`, 2015.

[189] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[190] Statista. Mobile Internet – Statistics & Facts. `http://www.statista.com/topics/779/mobile-internet/`, 2015.

[191] G. Suarez-Tangil and G. Stringhini. Eight Years of Rider Measurement in the Android Malware Ecosystem: Evolution and Lessons Learned. *arXiv preprint arXiv:1801.08115*, 2018.

[192] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[193] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security)*, 2009.

[194] P. Syverson, R. Dingledine, and N. Mathewson. Tor: The Second Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium (USENIX Security)*, 2004.

[195] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro. The Evolution of Android Malware and Android Analysis Techniques. *ACM Computing Surveys (CSUR)*, 49(4):76, 2017.

[196] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[197] M. Y. Tee and M. Zhang. Hidden App Malware Found on Google Play. `https://www.symantec.com/blogs/threat-intelligence/hidden-app-malware-google-play`, 2018.

[198] Telegram. FAQ for the Technically Inclined. `https://core.telegram.org/techfaq#q-do-you-have-forward-secrecy`, 2015.

[199] S. D. Tetali. Keeping 2 Billion Android Devices Safe with Machine Learning. `https://android-developers.googleblog.com/2018/05/keeping-2-billion-android-devices-safe.html`, 2018.

[200] P. Teufl, M. Ferk, A. Fitzek, D. Hein, S. Kraxberger, and C. Orthacker. Malware Detection by Applying Knowledge Discovery Processes to Application Metadata on the Android Market (Google Play). *Security and Communication Networks*, 9(5):389–419, 2016.

[201] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter. Resolving the Predicament of Android Custom Permissions. In *Proceedings of the Annual Symposium on Network and Distributed Systems Security (NDSS)*, 2018.

[202] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – A Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[203] M. Vanhoef and F. Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[204] R. Vasa, L. Hoon, K. Mouzakis, and A. Noguchi. A Preliminary Analysis of Mobile App User Reviews. In *Proceedings of the 24th Australian Computer-Human Interaction Conference*. ACM, 2012.

[205] D. Venkatesan. Android.Bankosy: All Ears on Voice Call-Based 2FA. `http://www.symantec.com/connect/blogs/androidbankosy-all-ears-voice-call-based-2fa`, 2016.

[206] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. *ACM SIGMETRICS Performance Evaluation Review*, 42(1):221–233, 2014.

[207] A. Villas-Boas. More than 500,000 People Downloaded Games on the Google Play Store that were Infected with Nasty Malware – Here are the 13 Apps Affected. https://www.businessinsider.com/google-play-store-game-apps-removed-malware-2018-11?r=US&IR=T, 2018.

[208] Violet Blue. Researchers Publish Snapchat Code Allowing Phone Number Matching After Exploit Disclosures Ignored. http://zd.net/1Qz8sRd, 2013.

[209] G. Wang, B. Wang, T. Wang, A. Nika, H. Zheng, and B. Y. Zhao. Whispers in the Dark: Analysis of An Anonymous Social Network. In *Proceedings of the 2014 ACM Internet Measurement Conference (IMC)*, 2014.

[210] X. Wei and M. Wolf. A Survey on HTTPS Implementation by Android Apps: Issues and Countermeasures. *Applied Computing and Informatics*, 13(2):101–117, 2017.

[211] Wickr. How Wickr Works. https://www.wickr.com/how-wickr-works/.

[212] M. Y. Wong and D. 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.

[213] B. Woods. Google Play has Hundreds of Android Apps that Contain Malware. http://www.trustedreviews.com/news/malware-apps-downloaded-google-play, 2016.

[214] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. DroidMat: Android Malware Detection Through Manifest and API Calls Tracing. In *Proceedings of the Asia Joint Conference on Information Security (AsiaJCIS)*, 2012.

[215] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective Real-Time Android Application Auditing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[216] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[217] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated Mining and Characterization of Fine-Grained Malicious Behaviors in Android Applications. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2014.

[218] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of the International Conference on Advances in Mobile Computing and Multimedia (MoMM)*, 2013.

[219] S. Yegulalp. Mozilla: It's HTTPS or Bust for Firefox. http://preview.tinyurl.com/kkahwud, 2015.

[220] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave Me Alone: App-Level Protection Against Runtime Information Gathering on Android. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[221] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life After App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[222] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[223] Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo. TruStore: Implementing a Trusted Store for Android. *DISI-Via Sommarive*, 2014.

[224] M. Zheng, M. Sun, and J. C. Lui. DroidTrace: A Ptrace Based Android Dynamic Analysis System with Forward Execution Capability. In *Proceedings of the IEEE International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2014.

[225] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the second ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[226] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[227] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2012.

Appendix A

SSL

A.1 Complete List of Apps Investigated

4shared	8 Ball Pool	Adobe AIR	Amazon
Amazon Local – Deals, Coupons	Amazon Music	Talking Angela	Angry Birds
Ask.fm - Social Q&A Network	AVG Antivirus Security – FREE	Badoo – Meet New People	BBM
Barclays Mobile Banking	Bible	Bitstrips	Booking.com Hotel Reservations
Calendar	Candy Crush Saga	Clean Master (Speed Booster)	Deezer Music
Dictionary.com	Dropbox	Google Drive	Duolingo: Learn Languages Free
Google Earth	eBay	Endomondo Running Cycling Walking	Facebook
Farm Heroes Saga	Flipagram	Flipboard: Your News Magazine	Fruit Ninja Free
Gmail	GO Launcher EX	GO Locker – Theme & Wallpaper	GO SMS Pro
Google+	Google Search	Groupon	Hangouts
Hill Climb Racing	Instagram	IMDb Movies & TV	Jetpack Joyride
Job Search	KakaoStory	KakaoTalk: Free Calls & Text	Keep
LINE: Free Calls & Messages	LinkedIn	Magic Piano by Smule	Magisto Video Editor & Maker
Maps	MeetMe: Chat & Meet New People	Facebook Messenger	Despicable Me (Minion Rush)
Netflix	Noom Coach: Weight Loss Plan	PayPal	PicsArt Photo Studio
Pic Collage	Pinterest	Play Movies & TV	Play Music
Play Newsstand	Play Store	POF Free Dating App	QR Droid Code Scanner
Shazam	Skype	Snapchat	SoundCloud – Music & Audio
SoundHound	Spotify Music	Subway Surfers	Talking Ben
Talking Tom 2	Tango: Free Video Calls & Text	The Simpsons: Tapped Out	Telegram
Temple Run 2	textPlus Free Text + Calls	TextSecure Private Messenger	Google Translate
TripAdvisor	Tumblr	TuneIn Radio	TweetCaster for Twitter
Twitter	VEVO	Viber	Vine
Voice Search	WhatsApp	WeChat	Words with Friends
Yahoo Mail	Yelp	YouTube	Zoosk

Table A.1: Complete list of apps examined in Chapter 4

A.2 SSL Pinning Using any of Two Keystores

```
public class PinManager implements X509TrustManager {

  /* Get a default keystore instance or specify your keystore type e.g
     ., BKS */
  KeyStore ks = KeyStore.getInstance(KeyStore. getDefaultType());

  private String keyAlgorithm;
  private PublicKey storedPubKey;
  private String serial;
  String alias = "myCert";

  public PinManager(InputStream file) throws KeyStoreException {
    try {
      ks.load(file , password);
```

```
    } catch (IOException e) {
        e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    } catch (CertificateException e) {
        e.printStackTrace();
    }
    /* Get certificate and associated public key */
    X509Certificate cert = (X509Certificate) ks.getCertificate(alias);
    storedPubKey = cert.getPublicKey();
    keyAlgorithm = storedPubKey.getAlgorithm();
    serial = cert.getSerialNumber().toString();
}

/* Create TrustManager with specified keystore */
try {
    TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
    tmf.init((KeyStore) myStore);
    for(TrustManager tm : tmf.getTrustManagers()) {
    ((X509TrustManager) tm).checkServerTrusted(chain,authType);}
}
catch(Exception e) {
    throw new CertificateException(e);
}

@Override
public void checkClientTrusted(X509Certificate[] x509Certificates,
    String authType) throws CertificateException {
    return; // The client is not authenticated but you can implement
        your own checks if desired
}


@Override
public void checkServerTrusted(X509Certificate[] x509Certificates,
    String authType) throws CertificateException {
    if(null == x509Certificates) { // throw error if cert chain is
        empty
        throw new IllegalArgumentException("Server X509Certificate array
            is null");
    }
    /* Get server certificate's public key */
    PublicKey pubKey = x509Certificates[0].getPublicKey();

    /* Check if keys match */
    final boolean keyMatch = pubKey.equals(storedPubKey);

    if(!(null != authType && authType.equalsIgnoreCase(keyAlgorithm)))
        { // throw error if public key algorithm is not same
```

```
      throw new CertificateException("Server AuthType does not match");
    }
    if (x509Certificates[0].getSerialNumber().toString() == serial)
      throw new CertificateException("Server serial number doesn't
          match");
    if(!keyMatch) {
      throw new CertificateException("Public key expected does not
          match");
    }
  }
}


/* One of the keystore should be commented... */

/* Development environment keystore */
int myStore = R.raw.devStore;

/* Production environment keystore */
int myStore = R.raw.proStore;

/* Initialize SSL context with customized TrustManager */
try {
  InputStream in = getResources().openRawResource(myStore); // create
      an input stream with specified keystore
  TrustManager trustManager[] = {new PinManager(in)};
  SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
  sslContext.init(null, trustManager, null);
  /* Create connection using customized socket factory*/
  URL url = new URL("https://www.bob.com/");
  HttpsURLConnection connection = (HttpsURLConnection) url.
      openConnection();
  urlConnection.setHostnameVerifier(hostnameVerifier);
  urlConnection.setSSLSocketFactory(sslContext.getSocketFactory());
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
} catch (KeyStoreException e) {
    e.printStackTrace();
} catch (KeyManagementException e) {
    e.printStackTrace();
}
```