IMPROVING MOBILE-MALWARE INVESTIGATIONS WITH STATIC AND DYNAMIC CODE ANALYSIS TECHNIQUES

Vom Fachbereich Informatik (FB 20)
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation von

Siegfried Rasthofer, M.Sc.

geboren in Landshut, Deutschland.

Referenten:
Prof. Dr. Eric Bodden (Referent)
Prof. Dr. Andreas Zeller (Korreferent)
Prof. Dr. Mira Mezini (Korreferentin)

Tag der Einreichung: 7. November 2016
Tag der Disputation: 22. Dezember 2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Darmstadt 2017
Hochschulkennziffer: D17

Siegfried Rasthofer:
Improving Mobile-Malware Investigations with Static and Dynamic Code Analysis Techniques

PHD REFEREES:
Prof. Dr. Eric Bodden
Prof. Dr. Andreas Zeller
Prof. Dr. Mira Mezini

FURTHER PHD COMMITTEE MEMBERS:
Prof. Dr. Reiner Hähnle
Prof. Dr. Christian Bischof
Prof. Dr. Patrick Eugster

Darmstadt, Germany January 2017

ABSTRACT

Similar to the PC world, the abundance of mobile malware has become a serious threat to smartphone users. Thousands of new apps or app versions are uploaded to popular app stores every day. All of them need to be analyzed against violations of the app store's content policy. In particular, one wishes to detect whether an application contains malicious behavior. Similarly, antivirus companies check thousands of apps every day to determine whether or not they are malicious. Both app store operators and antivirus vendors face the same problem: it is generally challenging to tell apart malware from benign applications. This is because malware developers aim to hide their applications' malicious behavior as long as possible from being detected by applying different obfuscation techniques. The raising sophistication with which such measures are implemented pose a serious problem not just to automated malware detection approaches but also to the manual analysis of potential malware by human experts.

In this dissertation, we propose a novel reverse engineering framework that includes different approaches for automatically extracting insights of the behavior of an Android application. In particular, we propose a novel approach, based on machine-learning, to automatically identify sensitive source and sink API methods. Furthermore, we propose an approach to automatically extract concrete runtime values, such as SMS messages or URLs, at any code location. The approach combines static and dynamic code analysis techniques in such a way that it is resistant against common obfuscation techniques. A further contribution is an approach that extracts concrete environment conditions that need to be fulfilled in order to reach a certain code location. This approach is based on code fuzzing that gets supported by static and dynamic code analysis techniques. All these approaches provide different insights into the analyzed application, in particular how and under which circumstances the application communicates with its environment.

The reliable extraction of these insights requires novel solutions that address fundamental limitations of current static and dynamic code analysis approaches. We, therefore, also contribute new code analysis techniques that reduce well-known limitations of code analysis, such as reflective method calls or inter-component communications, resulting in incomplete callgraphs, or complex path conditions that result in reachability problems.

All the insights that are extracted by our proposed approaches help human experts in speeding up their malware investigations. Manual malware investigations benefit from the automatic extraction of precise insights of the behavior of an application, which otherwise requires a time-consuming, manual analysis. On the other hand, existing automated code analysis approaches that are used during malware investigations benefit from our new techniques by reducing well-known limitations. This improves the detection rate of these approaches.

ZUSAMMENFASSUNG

Die Häufigkeit von mobiler Schadsoftware ist, ähnlich zur PC-Welt, ein großes Problem für Smartphone Benutzer geworden. Tausende von neuen Applikationen oder neuen Versionen von Applikationen werden täglich auf bekannte App Stores hochgeladen. All diese Applikationen müssen auf Verletzungen der App Store Richtlinien hin untersucht werden. Speziell werden hierbei Applikationen auf schadhaftes Verhalten untersucht. Ähnliches gilt bei Antiviren-Firmen, die täglich mehrere tausend Anwendungen auf schadhaftes Verhalten untersuchen müssen. Beide Parteien haben im Prinzip ein ähnliches Problem: es ist generell schwer zwischen gutartigen und schadhaften Applikationen zu unterscheiden. Gründe hierfür sind Entwickler von schadhaften Applikationen, die ihre Applikationen so programmieren, dass das schadhafte Verhalten so lange wie möglich unentdeckt bleibt. Dies wird durch unterschiedliche Verschleierungstechniken erreicht. Die Art der Verfahren wird jedoch immer komplexer und stellt somit nicht nur automatische Verfahren zur Erkennung von schadhaften Verhalten vor großen Herausforderungen, sondern auch manuelle Untersuchungen durch Experten.

In dieser Dissertation stellen wir ein neues Reverse Engineering Framework vor, welches unterschiedliche Verfahren zur automatischen Extraktion von sicherheitsrelevanten Informationen aus Android Applikationen beinhaltet. Konkret stellen wir in dieser Arbeit ein Verfahren zur automatischen Extraktion von sicherheitsrelevanten Quellen- und Senken-APIs vor, welches auf Techniken des maschinellen Lernens beruht. Des Weiteren wird ein Verfahren vorgestellt, welches vollautomatisch Laufzeitwerte an beliebigen Codestellen extrahiert. Dies könnten zum Beispiel konkrete SMS Nachrichten oder URLs sein. Das Verfahren kombiniert statische und dynamische Codeanalyse-Techniken so miteinander, dass es resistent gegen gängige Verschleierungstechniken ist. Ein weiteres Verfahren extrahiert konkrete Umgebungsbedingungen, die erfüllt sein müssen, um eine bestimmte Codestelle zu erreichen. Dieses Verfahren basiert auf Code-Fuzzing und verwendet statische und dynamische Codeanalyse Techniken. All diese unterschiedlichen Verfahren extrahieren unterschiedliche, sicherheitsrelevante Informationen aus einer Applikation, die Aufschluss darüber geben, wie und unter welchen Bedingungen eine Applikation mit der Umgebung interagiert.

Das Extrahieren dieser sicherheitsrelevanten Informationen bedurfte neuartiger Lösungen, die fundamentale Limitierungen von statischen und dynamischen Analysen lösten. Aus diesem Grund sind in dieser Dissertation neue Techniken beschrieben, die diese fundamentalen Limitierungen reduzieren. Dies beinhaltet neue Verfahren zur Verbesserung der Konstruktion von Aufrufgraphen, welche durch reflektive Aufrufe oder der Abbildung von Komponenten-Kommunikationen in Android erschwert wurden. Des Weiteren liefert diese Arbeit neue Techniken die sich mit dem Thema der Erreichbarkeit von Codestellen beschäftigt und neue Lösungsvorschläge aufzeigt.

Die in dieser Dissertation vorgestellten, neuartigen Verfahren helfen Analysten bei ihrer täglichen Arbeit in der Identifizierung von schadhaften Applikationen. Durch die automatische Extraktion von detaillierteren, sicherheitsrelevanten Informationen einer Applikation wird die manuelle Analysezeit einer Applikation essenziell reduziert. Bereits bestehende Werkzeuge, die von Analysten benutzt werden, profitieren ebenfalls von den in dieser Ar-

beit vorgestellten Verfahren. Dies ist auf die Reduzierung von statischen und dynamischen Limitierungen zurückzuführen. Somit können dem Analysten präzisere Ergebnisse vorgelegt werden, welche ebenfalls die Analysezeit reduzieren.

ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor Prof. Eric Bodden for his strong support, feedback and guidance. I am very grateful that Eric supported me as much as possible in all aspects of my academic carrier. He offered me numerous opportunities during my four years, which very positively influenced my personality and my carrier.

Besides my advisor Eric, I would also like to thank my colleagues of Eric's former chair in Darmstadt and my colleagues at Fraunhofer SIT. Especially the reading groups gave new impulses for different research directions, which resulted in publications like SuSi. I would particularly thank Steven Arzt and Stephan Huber. Steven was my office mate during the four years. Our brainstorming sessions, discussions and collaborations contributed to many research papers and also into a new commercial product, CodeInspect. Many thanks also to my Bavarian friend Stephan for our technical discussions about Android security and our hacking sessions. Both positively influenced many design decisions of HARVESTER and FuzzDroid.

I have been very lucky to work together with several excellent researchers. In particular, I would like to thank Michael Pradel from TU Darmstadt, who worked with me on the FuzzDroid project. I would also like to say thank you to Ben Livshits from Microsoft Research in Redmond for working with him during a summer internship. Additionally, I would also like to thank - alphabetically ordered - the following list of researchers for our collaborative work: Alessandra Gorla, Alexander Roßnagel, Andreas Zeller, Damien Octeau, Enrico Lovat, Jacques Klein, Konstantin Kuznetsov, Li Li, Mike Papadakis, Nicole Eling, Patrick McDaniel, Peter Buxmann, Philipp Richter, Tegawendé Bissyande, Vitalii Avdiienko, Voker Stolz, Yves le Traon. It was great working with you.

I had the privilege to supervise and work with several outstanding students. Max Kolhagen and Robert Hahn (each bachelor thesis), Dieter Hofmann and Julien Hachenberger (each master thesis). A special thank goes to Marc Miltenberger for his outstanding implementation support of different projects. It was a lot of fun and I am very grateful for your hard work.

Special thanks goes to Irfan Asrar, Carlos Castillo and Alex Hinchliffe from McAfee research lab in Santa Clara for your support of sharing different malware samples, our technical discussions and especially for giving me the chance to work with you on different malware investigations.

Most importantly, I would like to say thank you to my parents for supporting me during my study and research time and for always encouraging me in hard times. Without you, this would not have been possible.

Last but not least, I am also deeply thankful to my girlfriend for her assistance and patience in stressful days when workload was heavy.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Smartphones are commonplace today. The number of smartphone users worldwide is forecast to grow from 3.4 billion in 2015 to around 6.4 billion in 2021 [Eri16]. This is a very big market, which is dominated by Google with its Android operating system [Fut15]. A decisive reason behind the boom in smartphones is the wide range of applications (short app), which can be easily downloaded from different app stores. Very recently, Google announced at the Google I/O developer conference 2016 that 65 billion apps were downloaded and installed via Google Play. This is a tremendous number, which shows the success of the application business. There is an application for almost every need: social network applications for communicating with friends; email or banking applications for business purpose; gaming applications during spare time, etc.

This increasing smartphone market is not only attractive for users, but also for malware authors. Their main purpose is to gain money from the victim, which is realized by different techniques, such as *blackmailing* (*ransomware* [Anda]), *phishing attacks* for stealing sensitive information [Ras+15b], sending *costly SMS messages* [Ras+16] or initiating *costly calls* [Anda]. These kinds of techniques are implemented in mobile applications and distributed via different channels like app stores, mail attachments or via SMS messages containing a download link of the malware. Different statistics show that mobile malware applications are mostly distributed via Android applications [LLC15]. One reason therefore is probably Google's lead in the mobile market share.

In comparison to PC, smartphones are a more valuable and generic attack surface since they offer a lot more sensitive data, which can be easily accessed with simple API methods. More concrete, smartphones contain data such as phone numbers, call frequency, call history, SMS data, location data or professional and personal schedules that are accessible with simple API calls. In comparison, PCs do usually not contain such sensitive data (e.g., call history or SMS data) or do not provide a generic API for accessing this data directly. Instead, this data is distributed on the file system and the location heavily depends on the program that processes that data (e.g., Microsoft Office address book is stored at a different location as Thunderbird's).

One of the first discovered Android malware, FakePlayer, was detected in 2010 [Dun+14]. It was a trojan horse that attempts to send premium SMS messages to a hardcoded number once the user opens the malicious application. It was very easy to automatically or manually spot the malicious behavior since there was no obfuscation technique involved. However, during the years, malware authors applied more and more sophisticated obfuscation techniques into their malicious applications for hiding the malicious behavior as long as possible. For instance, in 2013 a malware named Obad [TKG13] appeared, which started obfuscating the malicious code with techniques like string encryption in combination with reflective method calls. This makes it very hard for a human malware analyst, but also for static code analysis approaches, to understand the code functionality. These approaches do not have a complete view of the application's code and, therefore, do not work as intended. In 2015, we detected a new banking trojan malware family called *Android/BadAccents* [Ras+15b], which contained different malicious components. The components, e.g.,

a phishing attack, were implemented in such a way that it is very hard, if not impossible with current analysis techniques, to automatically trigger (dynamic analysis) or detect (static analysis) the malicious behavior.

One of the reasons for the malware evolution towards adding more obfuscation techniques into the application is because of the improvement of automatic malware detection approaches by antivirus companies or the app stores itself. For instance, Google Play is protecting their users with Bouncer, a malware detection approach, which combines static and dynamic code analysis approaches. While Oberheide and Miller [OM12b] have shown that Bouncer performed only simplistic security checks on the application in 2012, Google improved Bouncer over the years resulting in a much higher detection rate of malicious applications [Anda]. However, even the current version of Bouncer does not detect every malicious application in the Google Play Store as different reports show [Rui16; Web16; PB15; PK16; Chy15]. For instance, in 2015 the AVAST security team detected a new malware family [Chy15] that affected millions of users and was distributed via the Google Play Store. Before an application gets uploaded and even while the application remains in the Google Play Store, it will be scanned for malicious behavior with machine-learning techniques that get supported from state-of-the-art static and dynamic code analysis approaches [Anda]. However, this particular malware might have not been detected by Google's system due to a simple *timing bomb*, a well-known limitation of dynamic code analysis approaches. The malicious application acted in a benign way until a certain point in time (e.g., after 30 days) when the malicious behavior started.

Timing bombs and other limitations pose severe research challenges for current code analysis approaches. In this thesis, we will cover some of those challenges and introduce new code analysis approaches that will improve the current state-of-the-art that address those challenges.

1.1 MOTIVATION

In the following, we first explain the major limitations of current static and dynamic code analysis approaches for automatically extracting insights about the behavior of an application (Section 1.1.1). Then, we summarize the major obfuscation techniques addressed in this dissertation in Section 1.1.2.

1.1.1 *Motivation Example*

In the following, we will focus on a concrete example showcasing that modern Android malware does not only pose a challenge for human analysts, but also for automated static and dynamic code analysis approaches. This main motivating example will be re-used in the following chapters and subsections for explaining the corresponding approaches. This example was inspired by our malware investigation together with Intel Security [Ras+15a; Ras+15b] in 2015. We identified a new threat campaign underway in South Korea that distributed a new form of Android banking trojan we designated as *Android/BadAccents*. By the time we had stopped the threat, within two months the trojan had infected more than 20,000 devices.

In that case, it was already known that we discovered a new form of Android malware, but concrete insights (e.g., *what data is sent where and when*) remained unanswered. Answer-

ing these questions needed a detailed manual reverse engineering approach, which was time consuming and, therefore, also cost-intensive.

Figure 1 shows a pseudo-code snippet, inspired by the *Android/BadAccents* malware[1] and the sophisticated malware families Obad[2], MobWin[3] and Tascudap[4]. Note that some methods are not implemented since the focus in this snipped is more on the different code analysis challenges instead of a complete example. It illustrates problems why automatic static and dynamic code analysis approaches did not help at that time. It shows a code snipped that leaks personal information (access to location information in line 16-18) to an email account (line 27). One cannot directly see the leakage since the `sendEmail()` method call was obfuscated via reflection. The credentials for the email account are implemented in native C/C++ code (line 2 and line 3). Both are accessed in the Java part (line 9 and line 10) and passed into the obfuscated `sendEmail()` method call. Those are very common obfuscation techniques in current malicious applications [TKG13; Ras+15b]. It makes it hard for a human analyst to understand the malicious behavior, but also for current static analysis approaches that do not consider reflective method calls or native C/C++ code. Further, an additional anti-analysis technique, a so-called emulator check in line 13 (`isEmulator()`), prohibits the execution of the data leak when the application runs on an emulator instead of a real device. Malware usually gets dynamically analyzed in a sandbox [Lin+14; Spr+13] by running in an emulator. Malware developers try to evade this by adding emulator checks into their code. The code also contains a second malicious behavior, which is implemented in line 32-51: a classical spam distribution. The malware sends spam messages to all contacts on the smartphone. Most of the time this technique is used for malware distribution, where SMS messages like "Hey, check out this awesome app www.malicious.com/malware.apk" are sent to all friends (contacts). Since it is sent to all personal contacts, it is more likely that a contact clicks on the link and downloads the malware. The text of the spam is received from an SMS-based C&C (Command and Control) communication with the attacker. This makes it very flexible changing the spam text. In line 38, the code shows a timing bomb, which stops the application for 30 minutes until it executes the malicious behavior. This kind of behavior targets dynamic analysis approaches that try to dynamically analyze the malicious application. Since the application waits for 30 minutes, the dynamic analysis has to wait that time before it can analyze the subsequent code. Additionally, in line 41, the application dynamically loads a new Android binary file called Dalvik Executable (dex). The name `"anserverb.db"` is used for confusion purpose, there is no database file loaded, instead the dex-file is just renamed. Line 43 and 44 shows the calling of the method `onStart()`, which returns a Boolean value. Only if the Boolean value is **`false`** (line 45), the malware continues and iterates over all contacts (line 46) and accesses the contact's phone number (line 47). In line 51, the malware sends a spam message via SMS message to all phone numbers of the contacts. The text of the spam is received via an incoming SMS message, which is sent by the attacker (C&C communication). The dynamic code loading part, together with the reflective method call is used to confuse the manual analyst and to add a barrier for static code analyses.

As a summary, the example contains two malicious parts, a data leakage in line 27 and a spam distribution in line 51. As we detail in the following, identifying these two malicious

1 Sample-MD5: a5028fd5df93ba753d919f02b7bf1106
2 Sample-MD5: 58617e6a483f59bc93e500c65116eb87
3 Sample-MD5: acfbb28b997275c365324e004c59771f
4 Sample-MD5: 7f12ec7d2fe0008085693 4b9da396261

```
1  class SMSReceiver extends BroadcastReceiver {
2  private native String getAccountName();
3  private native String getAccountPassword();
4
5  void onReceive(SMS sms, Intent intent) {
6
7    String body = sms.getSMSBody();
8
9    String accountName = getAccountName();
10   String accountPassword = getAccountPassword();
11
12   //emulator-check
13   if(!isEmulator()) {
14
15     //stores country information
16     String countryInfo = simCountryIso().equals("US") ? US : INTERN;
17     int cellID = CellLocation.getCid();
18     int lac = CellLocation.getLac();
19     String emailBody = countryInfo + " : " + cellID + " : " + lac;
20
21     //class: MailSender class
22     Class clazz = Class.forName(decrypt("1234", "ai03…")).getClass();
23     //method: sendEmail
24     Method method = clazz.getMethod(decrypt("1234", "fahg29favjvajii"), String.class, String.class,
         String.class);
25
26     //MailSender.sendEmail(emailBody, account-name, account-password)
27     method.invoke(emailBody, accountName, accountPassword);
28
29   }
30
31   //expects "#s:<spam message>"
32   if(body.startsWith("#s:") {
33     String spamMessage = body.substring(2);
34
35     Set<Contact> contacts = getAllContacts();
36
37     //30 minutes
38     wait(30 * 60000);
39     // dynamic class loading, expects a dex file,
40     // even though file suffix is .db
41     DexClassLoader dcl = new DexClassLoader("anserverb.db");
42     Class clazz = dcl.loadClass("BaseABroadcastReceiver");
43     Method method = clazz.getMethod("onStart", Intent.class);
44     boolean returnValue = (boolean)method.invoke(intent);
45     if (returnValue == false) {
46       for(Contact contact : contacts) {
47         String contactNumber = contact.getNumber();
48         //sends a spam message like "check out the following app:
49         //www.malicious.com/malware.apk" to all contacts
50
51         sendTextMessage(contactNumber, spamMessage);
52
53       }
54     }
55   }
56 }
57 }
```

Runtime Value Extraction
Environment Inference
Sensitive API Calls

Figure 1: Main motivating example: example inspired by the *Android/BadAccents*, Obad, MobWin and Tascudap malware that leaks location information via email and sends spam messages to contacts under certain circumstances

parts in a fully automatic way challenges current state-of-the-art static and dynamic code analysis approaches:

- **Identification of Sensitive API calls:** To get a first idea of the behavior of an application, it is important to analyze the usage of API method calls. This gives a manual reverse engineer a good overview about the general behavior. However, Android version 4.2 comprises more than 110,000 public methods [RAB14], it is an error-prone and time consuming undertaking to identify sensitive method calls. Even worse, every new major release of Android might introduce new sensitive API method calls. Sensitive API calls are not only important for a manual analysis, they are also mandatory input for many code analysis approaches for security purposes. Concrete examples are the detection of code vulnerabilities [Ege+13], the detection of malicious applications [Cha+13; Gor+14] or the detection of privacy leaks [Arz+14b]. The latter can be applied in the motivating example. This example contains less common API methods (line 17 and line 18), whose return values are sent to an email account. The sending method is obfuscated (line 27) by the usage of a reflective method call. However, if the dataflow tracking tool does not consider less common API calls, this data leak cannot be identified. Therefore, individual code analysis approaches can be as precise as possible, if their input - sensitive API methods - are not well defined, the tools may show a low recall, i.e., miss data leaks.

- **Runtime Value Extraction:** During a malware investigation, an analyst needs to extract concrete information about an application. This can include information about URLs the application is communicating with or the content of SMS messages, which is sent by the application. All these details are most of the time concrete runtime values that get passed into an API call. As an example, in our investigation we were interested in the username and password (line 9 and line 10) of the email account since we assumed that the attacker uses this channel for data leakage reasons.

  The research area of *runtime value extraction* has also a significant importance in the context of resolving reflective method calls. The usage of reflective method calls is a well-known limitation of static code analysis approaches in Java since either the analysis does not have a complete view of the code (missing callgraph edges) or it over-approximates in such a way that too many callgraph edges exist. However, if the concrete runtime values that get passed into the reflective method call are known, it is possible to identify the corresponding API call, which is essential for the construction of the callgraph. This "problem" is exploited in the context of code obfuscation as shown in Figure 1. Line 22 till line 27 and line 42 till line 44 show the usage of reflective method calls. This is not only a problem for static code analysis approaches, but also for manual code audits, which require a manual resolving of the API calls for a better code understanding.

  There are different approaches to extract runtime values from an application either in a purely static [Ege+13; Li+15a] (extraction of constant strings) or a purely dynamic way [Bod+11]. Both approaches fail once an application makes use of obfuscation techniques that target well-known limitations of static and dynamic approaches. Unfortunately, malware applications make use of these techniques.

- **Environment Inference:** To avoid being detected as malware through automated or manual analyses, many malware apps exhibit their maliciousness only when being

executed in a particular environment. For example, some apps check whether they are running in an emulator or another analysis environment, and behave benignly in these cases. Other malware apps target specific countries and remain harmless unless the SIM card in the victim's phone is registered in one of the target countries. Yet another kind of malware targets phones with a specific app installed, such as a vulnerable banking app.

To decide whether an application is malicious or not it is essential to infer such environment dependencies. Figure 1 for example shows a sensitive API call in line 51, the sending of a text message. This can be a benign behavior in the case of a messaging application or a malicious behavior in cases of SMS trojans or spam. However, determining the required environment setup for the application could provide important information that can ease the decision. More concretely, in our example, the SMS message is only sent if a well-formed SMS is received, the application needs to be opened for at least 30 minutes, there has to be a dex-file (`"anserverb.db"`) on the file system and there has to be at least one contact on the smartphone. Only then, an SMS message is sent to all contacts on the smartphone. These environment dependencies are very uncommon for benign applications and provide a major indication that the application is very likely to be malicious.

Static symbolic code analysis approaches [Fra+16; Sax+10] are a very common practice once one need to identify a constraint under which a certain code location gets reached. However, apart from the well-known limitations of static analysis approaches, another challenge that these approaches need to face is how to effectively handle the exponential number of path in checked code [BCE08; Kol+12a]. This results most of the time in performance penalties [CDE08b; Cad+08; GLM08; Sax+10]. Dynamic approaches such as concolic code execution [Seno7; CDE08b; Cha+12] or program fuzzing [MFS90; GLM12] need to face the well-known limitations of dynamic code execution.

1.1.2 *Common Obfuscation Techniques*

In the following, we summarize the most common obfuscation techniques in modern Android malware applications that complicate the extraction of security-relevant information with static and dynamic code analysis techniques. The techniques are selected based on related work [Hac16], our own malware investigations [Ras+15b] and personal feedback from an antivirus company. Modern Android malware applications mostly contain techniques such as:

STRING ENCRYPTION: Encrypting constant strings, such as concrete URLs, mostly preventing static analysis approaches from extracting concrete insights.

REFLECTIVE METHOD CALLS: Invoking a sensitive API call, such as `sendTextMessage` with a reflective method call, mostly prevents static analysis approaches from creating precise callgraphs, resulting in imprecise information extraction, e.g., too many false-positives or false-negatives in static dataflow approaches.

DYNAMIC CODE LOADING: Dynamic loading prevents static analysis approaches from analyzing the whole application's code.

TIME BOMBS: Actively adding delayed executions to an application usually negatively influences the dynamic extraction of security-relevant information, e.g., timeout before detecting the sending of a premium SMS message.

LOGIC BOMBS: Actively adding conditions that influence the execution of a certain malicious behavior, negatively influences dynamic analysis approaches. Examples are checks whether the application runs on an emulator or not.

INTEGRITY CHECK: Verifying whether the code of an application got modified, i.e., integrity violation, is usually applied to prevent an application from dynamic analysis techniques that make use of bytecode instrumentation techniques.

A more detailed explanation of these techniques will be described in Section 2.3.

1.2 GOAL AND SCOPE OF THIS DISSERTATION

The global goal of this dissertation is to aid the investigation of modern Android malware. The proposed techniques operate at the application layer of the Android operating system and the focus is on automatically extracting insights about the behavior of an application realized with code analysis approaches. More concretely, we will show that (1) current static and dynamic code analysis approaches that try to extract a certain behavior of an application, e.g., *what* data is sent *where*, do have different limitations. This prevents current approaches from extracting the behavior from applications. (2) We also improve the current situation by designing new code analysis techniques that help existing approaches for extracting application insights in performing much better results with current applications. (3) In addition to that, we also implemented new code analysis techniques and tools that provide more concrete details about the behavior of Android applications.

Providing concrete insights about the behavior of malicious applications is a very important step for manual malware analysts who have to decide if an application is malicious or not. However, it is also very important for a further decision on automatic approaches that try to automatically argue about the detection of malicious applications [Gor+14; Avd+15; Sha+12; Tam+15; BZNT11; Arp+14; Cha+13; AA15]. In this dissertation, we only focus on the first step, the extraction of concrete insights of an Android application and we do not automatically argue whether an application is malicious or not.

1.3 THESIS STATEMENT

This dissertation confirms the thesis that

> It is possible to create a framework that automates the extraction of fundamental insights about the behavior of an Android application, even if that application uses obfuscation techniques.

More precisely,

TS-1: It is possible to automatically identify security- and privacy-relevant API methods that read (sources) and write (sinks) from/to resources from a large application framework, the Android framework.

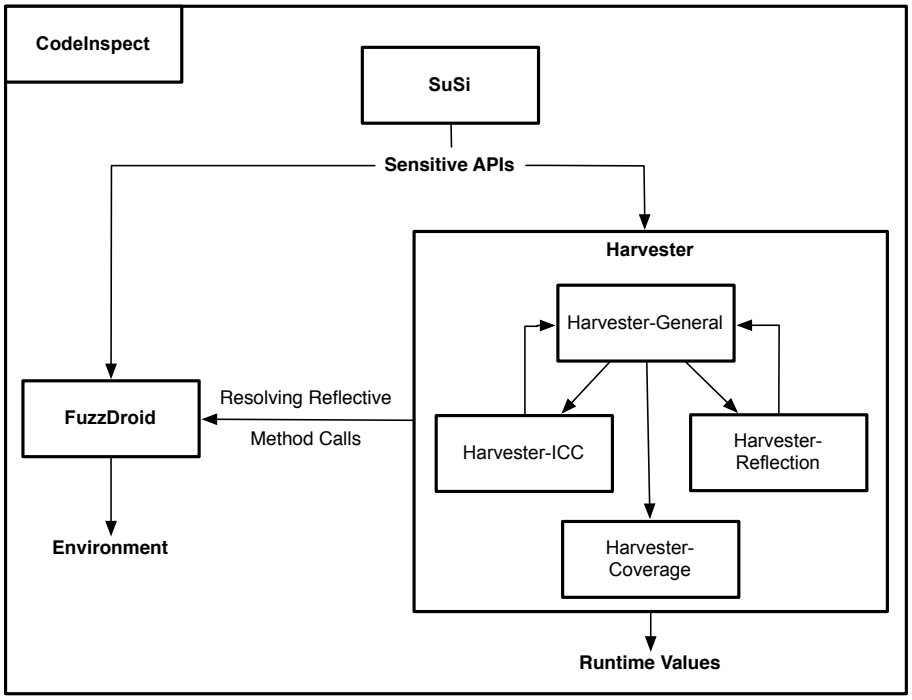

Figure 2: Main contributions and their relations

TS-2: It is possible to automatically extract runtime values at any code location in an Android app, even if it contains anti-analysis techniques mentioned in Section 1.1.2.

TS-3: It is possible to automatically resolve reflective method calls that are actively applied as an obfuscation technique for Android applications. This is even possible if the reflective method's parameter values are encrypted in a format that is not known beforehand.

TS-4 It is possible to create precise inter-component callgraphs (Android inter-component communications), even if the application is obfuscated with anti-analysis techniques mentioned in Section 1.1.2, which is essential for further static code analysis approaches that try to extract concrete insights about the application's behavior.

TS-5: It is possible to dynamically identify data leakages in an application, even if it is obfuscated with dynamic anti-analysis techniques described in Section 1.1.2.

TS-6: It is possible to automatically infer required environment-conditions under which a malicious application needs to run in order to reach a certain code location, even if the application contains anti-analysis techniques mentioned in Section 1.1.2.

1.4 SUMMARY OF CONTRIBUTIONS

Figure 2 gives a general overview of CODEINSPECT, a novel reverse-engineering framework for inspecting modern Android malware applications. In the following, we highlight the major contributions of this dissertation and how they relate to each other:

SUSI: IDENTIFICATION OF SENSITIVE API CALLS. We provide an automatic approach that identifies sensitive API method calls from the Android Open Source Project. More concretely, we automatically identify "Android Sources" that access sensitive resources, e.g., address-book data, and "Android Sinks" that write data to a resource, e.g., send text messages. Moreover, these API methods are automatically categorized into different categories, such as "unique-identifier" for certain sources or "SMS_MMS" for certain sinks. (Chapter 3)

HARVESTER: FULLY-AUTOMATIC EXTRACTION OF RUNTIME VALUES. HARVESTER is an approach that fully-automatically extracts runtime values that get passed into method invocations (values of arguments) during runtime. The proposed technique is even effective if the application is highly obfuscated (see Section 1.1.2). Besides the general functionality of extracting runtime values from Android applications, HARVESTER's technique improves current state-of-the-art limitations of static and dynamic code analysis approaches. We improved the construction of *static callgraphs* as well as improved the *recall of existing dynamic dataflow tracking approaches*. (Chapter 4)

As shown in Figure 2, HARVESTER takes the extracted sensitive API methods from SuSi as input. If an application uses a sensitive API call, HARVESTER tries to extract runtime values for these API calls.

FUZZDROID: ENVIRONMENT-EXTRACTION OF MALICIOUS BEHAVIOR. FuzzDroid is a target-driven fuzzing framework, which aims to reach a certain code location in an application. The main goal is to extract environment dependencies under which a certain code position gets reached. (Chapter 5)

The fuzzing framework makes use of SuSi's list of sensitive API calls and HARVESTER. In the latter case, FuzzDroid benefits from HARVESTER's possibility of resolving reflective method calls.

CODEINSPECT: PRECISE AND FAST REVERSE ENGINEERING OF ANDROID APPLICATIONS. We provide a new Android reverse engineering framework that aims for a faster and more precise investigation of an Android malware sample. It gives a human malware analyst the possibility to read, change and enhance the application's bytecode, which is represented in a human readable type-based intermediate representation. Furthermore, it offers various plugins that provide the analyst a detailed security overview of an application. SuSi, HARVESTER and FuzzDroid are essential plugins for a faster and more precise investigation of an application. (Chapter 6)

1.5 LIST OF PUBLICATIONS

This dissertation is based on several previous publications and one bachelor thesis as listed below.

[Mil14] Marc Miltenberger. "Slicing-basierte String-Extraktion in Androidapplikationen." Bachelor Thesis. Germany: TU Darmstadt, Apr. 2014.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks." In: *2014 Network and Distributed System Security Symposium (NDSS)*. 2014.

[Ras+15] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. "How Current Android Malware Seeks to Evade Automated Code Analysis." In: *9th International Conference on Information Security Theory and Practice (WISTP'2015)*. 2015.

[Ras+16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques." In: *2016 Network and Distributed System Security Symposium (NDSS)*. San Diego, 2016.

[Ras+17] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. "Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments." In: *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. ACM, 2017.

The following lists further publications in the context of Android security that have been published by the author.

[ARB13] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. "Instrumenting Android and Java Applications as Easy as abc." In: *Runtime Verification (RV)*. Lecture Notes in Computer Science. Springer, 2013.

[Arz+13] Steven Arzt, Kevin Falzon, Andreas Follner, Siegfried Rasthofer, Eric Bodden, and Volker Stolz. "How Useful Are Existing Monitoring Languages for Securing Android Apps?" In: *Software Engineering (Workshops)*. 2013.

[Arz+14a] Steven Arzt, Stephan Huber, Siegfried Rasthofer, and Eric Bodden. "Denial-of-App Attack: Inhibiting the Installation of Android Apps on Stock Phones." In: *Proceedings of the Fourth ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. Nov. 2014.

[Arz+14b] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps." In: *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM. June 2014.

[Arz+15] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. "Using Targeted Symbolic Execution for Reducing False-positives in Dataflow Analysis." In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2015. ACM, 2015.

[Avd+15] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. "Mining Apps for Abnormal Usage of Sensitive Data." In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE 2015. ACM, 2015.

[Eli+16] Nicole Eling, Siegfried Rasthofer, Max Kolhagen, Eric Bodden, and Peter Buxmann. "Investigating Users Reaction to Fine-Grained Data Requests: A Market Experiment." In: *49th Hawaii International Conference on System Sciences (HICSS)*. IEEE Computer Society, 2016.

[HR16] Stephan Huber and Siegfried Rasthofer. "How to do it Wrong: Smartphone Antivirus and Security Applications Under Fire." In: *Def Con 24*. Aug. 2016.

[HRA16] Stephan Huber, Siegfried Rasthofer, and Steven Arzt. "(In-) Security of Smartphone AntiVirus and Security Apps." In: *VirusBulletin 2016*. Oct. 2016.

[Li+15] Li Li, Alexandre Bartel, Tegawende Bissyande, Jacques Yves Klein, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps." In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE 2015. ACM, 2015.

[Li+16] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. *Static Analysis of Android Apps: A Systematic Literature Review*. Tech. rep. University of Luxembourg, Fraunhofer SIT/TU Darmstadt, University of Wisconsin and Pennsylvania State University, Apr. 2016.

[RA15] Siegfried Rasthofer and Steven Arzt. "(In-)Security of Backend-As-A-Service Solutions." In: *Blackhat Europe*. Nov. 2015.

[RCH15] Siegfried Rasthofer, Carlos Castillo, and Alex Hichliffe. "We know what you did this Summer: Android Banking Trojan Exposing its Sins in the Cloud." In: *18th Association of Anti-virus Asia Researchers International Conference (AVAR) 2015*. Dec. 2015.

[Ras+14] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. "DroidForce: Enforcing Complex, Data-Centric, System-Wide Policies in Android." In: *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*. IEEE. Sept. 2014.

[Ras+15a] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. *An Investigation of the Android/BadAccents Malware which Exploits a new Android Tapjacking Attack*. Tech. rep. TU Darmstadt and McAfee Research Lab, Apr. 2015.

[Ras+15b] Siegfried Rasthofer, Steven Arzt, Max Kolhagen, Brian Pfretzschner, Stephan Huber, Eric Bodden, and Philipp Richter. "DroidSearch: A Powerful Search Engine for Android Applications." In: *2015 Science and Information Conference (SAI)*. 2015.

[Ras+16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. "Reverse Engineering Android Apps With CodeInspect (invited paper)." In: *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security (IMPS)*. 2016.

[Ric+13] P. Richter, E. Bodden, S. Rasthofer, and A. Roßnagel. "Schutzmaßnahmen gegen datenschutzunfreundliche Smartphone-Apps - Technische Möglichkeiten und rechtliche Zulässigkeit des Selbstdatenschutzes bei Apps, DuD 2013." In: *DuD* (2013).

1.6 PRACTICAL SECURITY IMPACT

The contributions of this dissertation have had a large impact on the security setup of different companies and app developers. With the help of our newly developed CodeIn-

SPECT tool, we found two serious vulnerabilities in Google's AOSP (Android Open Source Project) [Arz+14a; Ras+15b]. One of them [Ras+15b] was unfortunately already exploited by attackers. Furthermore, we supported an antivirus company during a serious malware investigation [Ras+15b], which resulted in a takedown of a malicious server and a malicious email account. We stopped the distribution of the malware sample, which had already affected 20,000 people, and we extracted information aimed at identifying the malware authors. Moreover, HARVESTER supported us in discovering a major data leakage in Backend-as-a-Service solutions [RA15] due to improper protection mechanisms on the part of the application developers. With the help of Backend-as-a-Service providers such as Facebook or Amazon, we reached out to the developers and helped them fix the problem.

1.7 OUTLINE

In the following we will provide some background information about the topics discussed in this thesis (see Chapter 2). After that, we describe the SUSI approach in Chapter 3, the HARVESTER approach in Chapter 4, the FUZZDROID approach in Chapter 5 and the CODEINSPECT framework in Chapter 6. Discussions and some conclusions conclude this dissertation in Chapter 7.

2 BACKGROUND

In this chapter, we provide some basic background information that is important to understand the remainder of this dissertation. In particular, we focus on the topics of Android applications, Android malware and obfuscation techniques. As a first step, we introduce the Android architecture, along with a special focus on the application layer (Section 2.1). Section 2.2 covers all information about certain malicious applications that are addressed in this dissertation. Since modern malware makes use of code obfuscation, we introduce the most common obfuscation techniques in Section 2.3. We only list those techniques that we refer to in the remainder of this thesis.

2.1 INTRODUCTION TO THE ANDROID ARCHITECTURE

Android is an open-source, Linux-based software stack created for a wide array of devices [Andd]. The Android operating system is used in many different devices including smartphones, tablets, televisions or even cars. It consists of six major layers (ordered from the bottom to the top): the *Linux kernel*, the *hardware abstraction layer*, *Native C/C++ libraries* and the *Android Runtime*, the *Java API Framework* and the *System Apps* layer. The focus in this dissertation is mostly on the *Java API Framework* and *System Apps* layer. The former consists of different APIs written in Java that one needs to create Android apps. The API simplifies the reuse of core components. More details are provided in the next section. The latter one comes with a set of core apps for email, SMS messaging, calendars, Internet browsing, contacts, and more.

2.1.1 *Android Application*

Android apps are most of the time written in the Java programming language. The Java source code files are compiled into a register-based bytecode representation called Dalvik Executable (dex). The dex-file gets further packed into an archive called Android Application Package (APK) [Andc]. This APK not only includes the app's bytecode, it also contains further components such as the App Manifest, resource files, an assets folder and signature files [Andf]. The latter ones are available since Android requires all APKs to be digitally signed with a certificate before they can be installed on a device.

Every Android application consists of four different basic components, which will be described in more detail in the following:

ACTIVITY
: An Activity is an application component that contains a user interface. It can be used to click on a button, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface [Appa].

SERVICE
: In comparison to an activity, a service does not provide any user interface. It is used for long-running operations in the background such as playing music.

BROADCAST RECEIVER
: A broadcast receiver is an application component that is able to receive broadcast-messages from an app or the operating system. For instance, an application can declare a broadcast receiver for intercepting incoming SMS messages. Once the device receives an SMS, a specific callback gets called in the receiver class and the SMS message is intercepted.

CONTENT PROVIDER
: Content providers manage the access to a structured set of data. They are the standard interface that connects data in one process with code running in another process [Appd]. An example of a content provider is the interaction with a database such as the contact data.

The Android operating system is an event-driven system, which is reflected in the development of an application: every application and especially every component consists of a distinct lifecycle that must listen for changes in the application state and react accordingly [Mei12].

Almost all components are declared in the `AndroidManifest.xml` file that is part of the apk and contains all configuration information of an application. Broadcast receivers are the only component that can be created and registered programmatically during runtime [Appc]. All the other components are created automatically by the Android runtime.

2.1.2 *Inter Process Communication*

Android is a privilege-separated operating system. Every application runs with a distinct system identity, the Linux user ID and group ID in its own sandbox. Therefore, applications are isolated from each other and it is per default not possible, for instance, that one application accesses data from another application [Ande]. However, there are different ways how applications can communicate with each other. One of the most common one is the communication via messaging objects called *Intents*. Intents allow a communication between activities, services and broadcast and they are handled via messages that are sent between components.

An intent can be *explicit*, directly specifying the component name of the target where the message needs to be sent. This is usually realized by adding the concrete class name to the intent. However, there is also the possibility of *implicit intents* that only declare the functionality that is desired for the target component. In such cases, the desired functionality is described by the following fields [Appf]:

ACTION Specifies the generic action that need to be performed. For instance, if "VIEW" is specified, the sender of the intent wants some data do be displayed.

CATEGORY Describes additional information about the kind of component that should handle the intent. For instance, the category BROWSABLE indicates that the target component shall be a web browser to display information.

DATA References data to be acted on. This can, for instance, be a phone number or a host address.

More details about the individual items is described in related work [Appf].

Components that wish to receive an implicit intent need to define an *intent filter* that contains all attributes it is willing to receive. This includes action, category or data items. Furthermore, components have the additional attribute called *exported*, if set to *true* indicate that components in other applications are allowed to interact with this specific component. This allows inter-application communications. If the flag is set to *false*, only components within an application or applications with the same user ID can interact with the component. There is also the possibility to define permissions on intents that restrict the communication with specific components.

Matching an intent to a specific component or to different components is realized by the Android system during runtime. For *explicit intents*, the target component is directly addressed. Therefore, the Android system directly calls the target component. For implicit intents, the intent resolution process makes use of the action, category and data attributes of an intent to determine the target components. More concrete information about the intent resolution process is described online [Appf] and in related work [Oct+16].

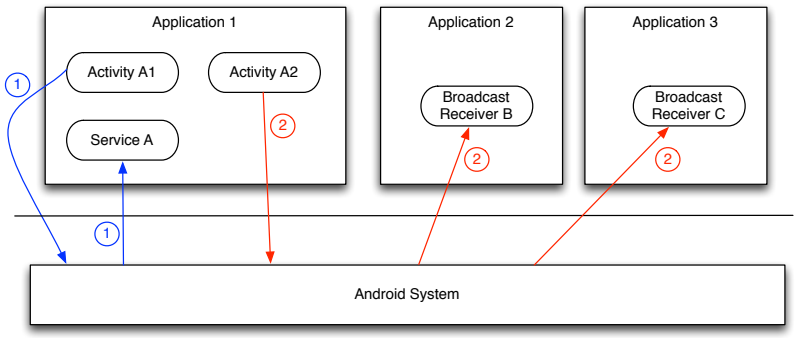

Figure 3: Intra- and Inter-Application communications

Figure 3 shows an example that explains the different communication flows for explicit and implicit intents. In case of an explicit intent (intent 1) within an application, the message first goes to the Android system, which directly calls the target component (Service A). For implicit intents (intent 2), the Android system matches the corresponding components (applications). This is realized by showing the user different applications that match for that certain intent and the user decides which application shall proceed with the message.

Analyzing intent-based communication can be crucial to understanding an app's behavior. Such an analysis is called *inter-component analysis* if two components are communicating with each other, independent of the fact whether two distinct apps are communicating or two components within a single application. An analysis that only considers a single component is called an *intra-component analysis*.

2.1.3 *Security Model*

The main goal of this dissertation is to extract insights of the behavior of an application, which can be used to identify malicious applications, either manually or with automated approaches. Since Android provides different security features to protect the user against malicious behavior, it is worth discussing the individual features in the following. We will focus only on the main security features that try to prevent the user from installing malicious applications or that protect the user once a malicious application is installed.

Cloud-Based Security Analysis

Before developers publish their applications in the Google Play Store, every application undergoes a review process that checks whether the application violates the developer content policy. Examples for violations are restricted content (e.g., sexually explicit content) or malicious behavior. This process is also applied to applications that are already released in the app market. It is based on an automated machine-learning approach that makes use of static and dynamic code analysis techniques for identifying policy violations [Anda]. It further involves also manual inspection where automated analysis was not precise enough [Anda]. If the system detects a violation, it will remove the application from the Play Store and attempt to remove the application from all users' devices, using the *Verify Apps* protection mechanism (see next section).

On-Device Protection

In Android 4.2, Google introduced a new on-device protection feature called *Verify Apps*. It extracts different fingerprints and heuristics from an application and determines based on the Google database whether a particular application is already known as malware or whether an application contains malicious behavior [Dra+14]. Verify Apps can prompt the user to remove a potentially malicious application, can remotely delete it or can even block it from installation [Anda].

Android Platform Security

The Android operating system itself also consists of different security features that protect the user against malicious applications.

SECURITY SETTINGS Different security settings [Devb] protect the user's privacy and security. Examples are settings that prevent app installations from other sources than Google Play. Most of the time, malware is distributed not through Google Play but via third-party stores, via spam or social engineering by directly sending an APK to the victim. This, however, requires the user to enable the security setting (prevent non-market-apps) and should raise suspicions.

PREVENTING COST-SENSITIVE ACTIONS A cost-sensitive action is any action that might generate costs to the user or the network [Appb]. Examples are premium SMS messages or premium phone calls. Android tries to prevent applications from sending premium SMS messages by checking the sender number against well-known premium numbers before the SMS message is being sent. When addressing an SMS to such numbers, the user gets prompted with a dialog with information about potential SMS Fraud that needs to be confirmed by the user before the SMS gets sent.

ANDROID PERMISSIONS As a security measure, Android applications can only access sensitive API calls if a certain permission is defined in the application. As an example, if the developer wants to send an SMS message, she needs to request the `SEND_SMS` permission. If this is not the case, once the SMS API gets called during runtime, a security exception is thrown and the application terminates. During installation, the user prompts a window containing all permissions that the application requests. To install the application, one needs to grant them all. This all-or-nothing solution was improved upon by Android 6,

which introduced a more granular way of revoking and granting permissions. Android 6 allows the user to decide which permissions she wants to grant for a particular application and is able to dynamically grant permissions once the application accesses an API method that requires a permission [Anda].

2.2 MOBILE MALWARE

Throughout this dissertation, we often refer to the term *malware investigation*. A malware investigation is meant to be an investigation that verifies whether a certain application is malicious or not. This can be realized by an automated approach that takes arbitrary applications as input and automatically decides whether the application is malicious. The alternative is a semi-automated approach that automatically reports different security-relevant findings, which then need to be verified by a human analyst. Furthermore, a malware investigation can also be a manual analysis (human reverse engineering) of a potential malicious application or a manual analysis of a known malicious application. In the latter case, an analyst usually needs to extract specific information, e.g., email account credentials that is required for a further investigation, e.g., botnet takedown.

During such malware investigations, it is sometimes necessary to spot different malicious behaviors inside an application. In the following we describe different kinds of malicious behavior, which is referred to in this dissertation.

Financial Fraud

There are different ways for mobile applications to commit financial fraud. SMS Fraud [Anda] that charges the victim for costly outgoing SMS messages or call fraud [Anda] for making costly calls, are the two common ones. In this thesis, we will often focus on SMS Fraud. Android's countermeasure of providing a dialog once a premium SMS is sent by an application is offered since Android 4.2. However, it is only a blacklisting approach that is able to identify premium SMS numbers with blacklisting patterns (e.g., numbers containing four digits or less might be premium SMS numbers). Furthermore, tapjacking attacks [Yin+16] can be used to hide the security dialog from the user by providing a window that overlaps the security dialog. This hides the malicious activity from the user.

Mobile Phishing Attacks

In traditional phishing attacks, an attacker tries to draw a victim to a rogue website by sending her an email containing a link she is lured into clicking on [DT05]. Once done, the rogue website usually looks exactly like a known website but the website serves only to capture the user's personal information like passwords or credit card information. The same techniques are applied by mobile phishing attacks. The step of drawing the victim to a rogue website is in many cases realized by SMS Phishing [Sne16]. The victim gets an SMS message from a service or more commonly from an infected smartphone of a friend. In the latter case, the installed malware sends a Phishing SMS message iteratively to all contacts of the victim's address bock.

Personal Information Theft

In the case of personal information theft, one has to differentiate different aspects: One thing is information theft that is intended by the app developer. For instance, a restaurant finder app usually sends location data to a benign server. There are also many applications that include third-party advertisement libraries that send sensitive user data to their servers, because of their business model [Gra+12]. This kind of information theft is usually considered as *potential unwanted applications* (PUA) or *greyware*. But there is also information theft that steals sensitive user data for sending it to the attacker. This is an outright malicious behavior. Differentiating whether an information theft is malicious or not heavily relies on the context.

Command-And-Control

Traditional PC-based botnets usually consist of many infected PCs that receive commands from the command-and-control server. The communication hereby works with commonly used protocols like IRC or HTML for communicating with the infected PCs in a way that it is hard to detect by intrusion detections systems, for instance. The botnets have different goals. Spam distribution, initiating a distributed denial-of-service attack or stealing personal information are common examples [PO12]. Botnets in the mobile world, and especially for the Android OS, act very similar. However, many of these botnets make use of the SMS protocol as a communication channel since it is available on most of the devices. In this case, the attacker sends an SMS message to the infected device and the malicious application tries to intercept the incoming SMS message. The message itself contains different commands that are used for triggering malicious activities such as sending spam SMS messages to all contacts on the device (see Section 1.1.1).

2.3 BASIC OBFUSCATION TECHNIQUES

A common way of protecting the intellectual property of benign apps, even if sometimes questionable (security by obscurity), is to apply code obfuscation techniques. Malicious applications, on the other side, make use of code obfuscation for preventing the detection of the malicious behavior as long as possible. There are many different techniques in the area of code obfuscation and especially for obfuscating Android applications [Hac16]. In the following, we explain those techniques that are important for the remainder of this thesis.

CODE TRANSFORMATIONS Code transformation techniques are typically used for protecting the application from static analysis approaches including manual reverse engineering. Static code analysis approaches have different limitations [Ras+15b] that are exploited by obfuscators. *String encryption* is one of those techniques. It takes a constant string of the code, e.g., a URL, and encrypts it. The obfuscator further adds additional code statements that automatically decrypt the encrypted code during runtime. This means the constant string gets transformed in such a way that the value will be only available during execution. This makes it harder and more time-consuming for a human analyst to understand the behavior of an application, e.g., to which server the application is communicating with. Another common technique is the usage of reflective method calls. Reflective method calls provide a way of invoking methods by hiding the original method call into an `invoke()`

method [Ora]. The combination of string encryption together with reflective method calls make it very hard for static analyses to determine what method gets invoked, resulting in an incomplete callgraph of the application. However, this combination makes it also harder for a human analyst to understand the behavior of the application.

INTEGRITY VERIFICATION Protecting the integrity of an applications, e.g., the application's code, is important in different aspects. In the context of obfuscation, integrity checks are usually applied to detect application manipulations. This can be used for identifying dynamic malware analysis approaches that rely on application modifications, e.g., through the adding of logging information to the bytecode. In this case, the bytecode will be changed and the integrity verification check inside the application will fail, resulting in application termination or other unintended app behaviors. Since all Android applications need to be signed (see Section 2.1), a common way to implement an integrity check for Android applications is to bind the certificate to the application's code. This can be realized by checking whether the hash of the certificate that is used for app signing, has been changed. In case of a code manipulation, the application needs to be re-signed with a new certificate and therefore, the hash will be different.

LOGIC BOMBS Logic bombs cause an app to suppress any malicious activity when specified conditions are met [Chy15]. For example, the app itself detects that it is executed within an analysis environment [Spr+13; Lin+14]. In such cases, it behaves different than it would do on an infected device, since it usually gets analyzed for malicious behavior. The same applies for emulator checks, since most applications run on a real device. Testing an application on an emulator is usually applied for inspecting the application against malicious behavior. Further checks that are treated in this thesis are checks for rooted devices. An application on a rooted device has more privileges than on a non-rooted one. This results in the fact that on such a device a malicious application can access sensitive data from another application, which is not usually possible in the non-rooted (sandboxed) case.

There are different ways to implement logic bombs in Android applications. Many of them are covered in related work [Hac16].

TIME BOMBS Time bombs cause an app to suppress a specific behavior in any case for a longer period of time. This can be realized by a specific date when the behavior gets triggered, after a certain amount of time (e.g., after 30 minutes), or after certain events (e.g., a reboot of the phone), for instance. This technique is used by malicious applications to hide the malicious behavior from dynamic code analysis approaches such as sandboxes [Spr+13; Lin+14], which usually analyze an application for a limited amount of time.

PACKERS Packers are software programs that compress and encrypt other executable files in a disk and restore the original executable file when the packed files are loaded into memories [Rie+11; YZA08]. This kind of technique protects the application from static code analysis, including manual reverse engineering, since the main code is protected by encryption. Attackers apply packers to their malicious code for evading signature-based detection approaches as applied by many antivirus solutions [AN14].

3

IDENTIFICATION OF SENSITIVE SOURCES AND SINKS

CodeInspect is a framework comprising different plugins that provide various insights about the behavior of an application. These plugins are implemented by different kinds of code analyses. Many of these automated approaches, such as the dataflow tracking plugin (see Section 6.1.4), the automated extraction of runtime values (see Chapter 4) or the automated extraction of environment information (see Chapter 5) rely on knowledge of a complete set of sensitive API calls that are used within an application. If this list is not complete, it is very likely that these approaches are not able to extract important insights. Apart from the analyses within the CodeInspect framework there are also other approaches [Spr+13; Lin+14; RC15; Gor+14; Arp+14] that provide security-relevant information of an application, but rely on the identification of sensitive API methods. Besides automated approaches, also manual investigations benefit from information about sensitive API methods that are used within an application. It provides an analyst with a first impression about the different functionalities of the application, which may point to relevant code sections that need a deeper inspection.

Two classes of sensitive API calls are particularly important in security analyses: sensitive *sources* and sensitive *sinks*. Sources are API methods that access sensitive data such as location information or device information while sinks are a gateway to the outside of the smartphone like sending SMS messages or sending post requests via the Internet (a more detailed definition is provided in Section 3.3). This separation is especially important for dataflow analysis approaches, since they try to identify data leakages based on a connection between a source and a sink. In the following chapter we, therefore, mainly focus on the dataflow problem but we will also outline further application scenarios in Section 3.6 and in Chapter 6.

In version 4.2, Android's runtime library comprises about 110,000 public methods. This clearly makes a manual classification of sources and sinks infeasible. Furthermore, each new Android release includes new functionality (e.g., NFC in Android 2.3 or restricted profiles in Android 4.3), which often also leads to new sources and sinks. A manual identification of sensitive API methods is therefore impractical. It would impose a high workload on the analyst and would have to be done again for every new Android release. Additionally, handpicking is an error-prone task.

The fundamental research question addressed in this chapter is

> How can one automatically identify sensitive source and sink API methods from a large application framework, the Android framework?

We therefore propose SuSi, an automated machine-learning guided approach for identifying sources and sinks directly from the code of the Android runtime library. We have identified both semantic and syntactic features to train a model for sources and sinks on a small subset of hand-classified Android API methods. SuSi can then use this model to classify arbitrarily large numbers of previously unknown Android API methods.

Awareness of sources and sinks is highly useful but if a leak is found, the user often desires additional information on *what* information has leaked *where*, for instance location information to the Internet.

Therefore, we address a second research question in this chapter:

> How can one automatically categorize sensitive source and sink API methods from a large application framework, the Android framework?

SuSi thus further classifies the identified sources and sinks into source and sink categories. As we find, all categories contain more than a single method. The categorization hence shows that there is often more than one way to retrieve a certain piece of data, and that there are multiple ways to send it out to an attacker.

Contributions. To summarize, this chapter presents the following original contributions:

- a practical and precise definition of data sources and sinks in Android applications,
- an automated, machine-learning based approach for identifying data source and sink methods in the Android framework, even in case of new, previously unseen Android versions and variants,
- a classifier for data source and sink methods into semantic categories like network, files, contact data, etc., and
- a categorized list of sources and sinks for different Android versions, as well as the Google Mirror and Google Cast APIs. Existing static and dynamic analysis approaches can directly us the list.

Chapter Outline. In this chapter, we will first use a motivating example to describe the need for an automatic extraction of sensitive sources and sinks (Section 3.1). Then we will give some background on machine-learning for our purpose (Section 3.2), continue with some definition (Section 3.3) and explain the approach in Section 3.4. After that, we continue with the evaluation of our approach (Section 3.5), mention further application scenarios (Section 3.6) and show the limitations of the approach in Section 3.7. Related work in Section 3.8 and a summary (Section 3.9) conclude this chapter.

3.1 MOTIVATION AND CONTRIBUTION

As mentioned earlier, comprehensive lists of sources and sinks are hard to come by. As a consequence, lists of sources and sinks known from the scientific literature [Enc+10; Enc+11b; FCF09] only contain a few well-known Android API methods for obtaining and sending out potentially sensitive information. Section 3.5 gives detailed information about the current state-of-the-art. However, there are often multiple ways to achieve the same effect. Developers of malicious applications can thus choose less well-known sources and sinks to circumvent analysis tools. Let us assume an attacker is interested in obtaining the user's location information and writing it to a publicly accessible file on the internal storage without being noticed by existing program-analysis approaches.

```
1  void onCreate() {
2  //source: cell-ID
3  int cellID = CellLocation.getCid();
4  //source: location area code
5  int lac = CellLocation.getLac();
6  boolean darmstadt = (lac == 4222 && cellID == 44044);
7
8  String taint = "Darmstadt: " + darmstadt + " (" + cellID + " | " + lac + ")";
9  String f = this.getFilesDir() + "/mytaintedFile.txt";
10 //sink
11 FileUtils.stringToFile(f, taint);
12 //make file readable to everyone
13 Runtime.getRuntime().exec("chmod 666 "+f);
14 }
```

Listing 1: Motivating example SuSi: Android location leak example

Listing 1 shows an example that attempts to disguise a data leak by using less common methods for both the source and the sink. The example contains the same sources as in the introduction's motivation example (see Figure 1), the sinks are different however. More concrete, we have two source methods. Firstly, line 3 calls *getCid()*, returning the cell ID. Line 5 then calls *getLac()*, returning the location area code. Both pieces of data in combination can be used to uniquely identify the broadcast tower servicing the current GSM cell. While this is not an exact location, it nevertheless provides the approximate whereabouts of the user. In line 6 the code checks for a well-known cell-tower ID in Darmstadt, Germany. An actual malicious app would perform a lookup in a more comprehensive list. Finally, the code needs to make the data available to the attacker. The example creates a publicly accessible file on the phone's internal storage, which can be accessed by arbitrary other applications without requiring any permissions. Instead of employing Java's normal file writing functions, the code uses a little-known Android system function (line 11) which SuSi identifies as a "FILE" sink but which is normally hidden from the SDK (Software Development Kit): the *FileUtils.stringToFile* function can only be used if the application is compiled against a complete platform JAR file obtained from a real phone, as the *android.jar* file supplied with the Android SDK does not contain this method. Nevertheless, the example application runs on an unmodified stock Android phone.

This example is, at least for the source methods, a representative example for malware [Vir] we inspected. We have tested this example with publicly available static and dynamic taint analysis tools including Fortify SCA [Enc+11b], SCanDroid [FCF09], IBM AppScan [Appe] and TaintDroid [Enc+10] and confirmed that none of these tools detected the leak. This shows how important it is to generate a comprehensive list of sources and sinks for detecting malicious behavior in deceptive applications. SuSi discovers and classifies appropriately all sources and sinks used in the example.

3.2 BACKGROUND ON MACHINE LEARNING

SuSi uses *supervised learning* to train a classifier on a relatively small subset of manually-annotated training examples. This classifier is afterwards used to predict the class of an arbitrary number of previously unseen test examples. Classification is performed using a set of features. A feature is a function that associates a training or test example with a value, i.e., evaluates a certain single domain-specific criterion for the example. The approach

ID	Experience	Alcohol	Phone No	Accident
T1	5 yrs	0.6	1234	yes
T2	11 yrs	0.4	45646	yes
T3	3 yrs	0.2	76546	yes
T4	4 yrs	0.0	54645	no
T5	10 yrs	0.2	78354	no
C1	6 yrs	0.1	6585	?
C2	12 yrs	0.55	67856	?

Table 1: Classification example on drunk driving

assumes that for every class there is a significant correlation between the examples in the class and the values taken by the feature functions.

As a simple example, consider the problem of estimating the risk of a driving accident for an insurance company. We may identify three features: years of experience, blood alcohol level and the driver's phone number. Assume the learning algorithm deduces that a higher level of experience is negatively correlated with the accident rate, while the alcohol level is positively correlated and the phone number is completely unrelated. The impact of a single feature on the overall estimate is deduced from its value distribution over the annotated training set. If there are many examples with high-alcohol accidents, then this feature will be given a greater weighting than the years of experience. However, if there are more accidents of inexperienced drivers in the training set than alcohol-related issues, the classifier will rank the experience feature higher.

The classifier works on a matrix, organized with one column per feature and one row per instance. Table 1 shows some sample data. An additional column indicates the class and is only filled in for the training data. In our example, this column would indicate whether or not an accident took place. The first five rows are training data, the last two rows are test records to be classified.

In this example, a simple rule-based classifier would deduce that all reports with alcohol levels larger than 0.2 also contained accidents, so C2 would be classified as *accident:yes*. However, since the converse does not hold, further reasoning is required for C1. Taking the experience level into account (assuming that experienced drivers are drivers with more than 5 years of experience), there are two records (T3 and T4) of inexperienced drivers with levels of 0.2 or below in our test set: one with an accident and one without. In this case, the classifier would actually pick randomly, since both *accident:yes* and *accident:no* are equally likely. A probabilistic classifier could also choose *accident:yes* because accidents are more likely for inexperienced drivers (two out of three) in general. This demonstrates that results can differ depending on the choice of the classifier.

As a concrete classifier, we use *support vector machines* (SVM), a margin classifier, more precisely the *SMO* [Pla98] implementation in Weka [Hal+09] with a linear kernel. We optimize for minimal error. The basic principle of an SVM is to represent training examples of two classes (e.g., "sink" and "not a sink") using vectors in a vector space. The algorithm then tries to find a hyper-plane separating the examples. For a new, previously unseen test example, to determine its estimated class, it checks on which side of the hyper-plane it belongs. In general, problems can be transformed into higher-dimensional spaces if the data is not linearly separable, but this did not prove necessary for any one of our classification problems.

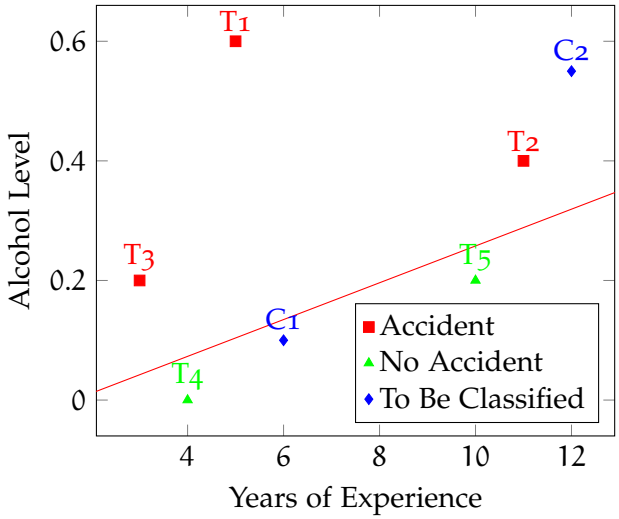

Figure 4: SMO classification example

Figure 4 shows an SMO diagram for Table 1. We have not included the phone number feature since it is unrelated to the probability of an accident. The red line shows a projection of the hyper-plane. In this example, the SMO detects that all points above the line are positive examples (i.e., records of accidents), and all points below are negative ones (i.e., no accident). C2 would thus be classified as an accident, just as with the simple rule-based classifier above, but C1 would now definitely be classified as non-accident because it lies below the line.

SMO is only capable of separating two classes. However, in SuSi, we have three classes in the first problem (source/sink/neither) and a lot more in the second one (the categorization). We solve the problem with a one-against-all classification, a standard technique in which every possible class is tested against all other classes packed together to find out whether the instance corresponds to the current single class or whether the classification must proceed recursively to decide between the remaining classes.

We also evaluated other classification algorithms based on different principles, for instance Weka's J48 rule learner, which implements a pruned C4.5 decision tree [Qui93]. The main problem with a rule set is its lack of flexibility. While many source-method names, for instance, start with *get*, this is not the case for *all* source methods. On the other hand, not all methods that start with *get* are actually sources. Since this rule of thumb is correct most of the time, however, a rule tree would usually include a rule mapping all *get* methods to sources and only perform further checks if the method name has a different prefix. With an SVM, such aspects that are usually correct, but not always, can be expressed more appropriately by shifting the hyper-plane used for separation.

Probabilistic learning algorithms like Naive Bayes [Zha04] produced very imprecise results. This happens because our classification problem is *almost* rule-based, i.e., has an almost fixed semantics. The variance is simply not large enough to justify the imprecision introduced by probabilistic approaches, which are rather susceptible to outliers.

3.3 DEFINITION OF SOURCES AND SINKS

Before one can infer sources and sinks, one requires a precise definition of the terms "source" and "sink". Several publications in the area of taint and information-flow anal-

ysis discuss sources and sinks, but all leave open the precise definitions of these terms. For instance, Enck et al. [Enc+10] define sinks informally as "data that leaves the system" which is, however, too imprecise to train a machine-learning based classifier; such classifiers are only as good as their training data.

Taint and information-flow analysis approaches track through the program the flow of *data*. Sources are where such dataflows enter the program and sinks are where they leave the program again. This requires us to first define *data* in the context of dataflows in Android applications.

Definition 1 (Data) *A piece of data is a value or a reference to a value.*

For instance, the IMEI in mobile applications is a piece of data, as would be the numerical value 42. We also treat as data, for instance, a database cursor pointing to a table of contact records, since it directly points to a value and is thus equivalent in terms of access control.

In taint tracking, one monitors the flow of data between resources such as the file system or network. Conversely, due to Android's app isolation, data that is simply stored in the app's address space is not of interest. Before one can define sources and sinks, one must therefore define the notion of a resource method. Mobile operating systems like Android enable applications to access resources using pre-defined methods. While one could also imagine fields being used for resource access, we found this not to be the case with Android.

Definition 2 (Resource Method) *A resource method reads data from or writes data to a shared resource.*

For instance, the operating system method for reading the IMEI (*getDeviceId()* in class *TelephonyManager*) is a resource method. In this case, the phone's hardware itself is the resource as the IMEI is branded into the silicon. The *sendTextMessage()* method in class *SmsManager* is a resource method for sending text messages to a specific phone number. The resource is the GSM network.

Note that a writing resource method does not necessarily need a reading counterpart. In our definition, there is no restriction on how the data is shared. A writing resource method might, for instance, send out data over the network (which is a resource). Though another application cannot directly obtain this data through a simple method call, the data can easily be sniffed from the network and is thus shared. Data leaving the phone is thus always considered shared.

After defining *data* and *resource methods* we can now define sources and sinks in the context of Android applications:

Definition 3 (Android Source) *Sources are calls into resource methods returning non-constant values into the application code.*

The *getDeviceId()* resource method is an Android source. It returns a value (the IMEI) into the application code. The IMEI is considered non-constant as the method returns a different value on every phone. Looking at the source code alone does not reveal this value. In contrast, a function that just reads a fixed constant from a database is a resource method but, by our definition, is not an Android source.

Note that our definition of sources does not make any restrictions on whether the data obtained from a source is actually *private*. SuSi will thus, at first, report sources of non-private data as well. However, in a second step SuSi then applies a further categorization which partitions sources into different categories of private data. This partitioning includes a class NO_CATEGORY, which represents sources of non-private data, which privacy-analysis tools can ignore. Details will be given in Section 3.4.1.

Definition 4 (Android Sinks) *Sinks are calls into resource methods accepting at least one non-constant data value from the application code as parameter, if and only if a new value is written or an existing one is overwritten on the resource.*

The *sendTextMessage()* resource method is an Android Sink as both the message text and the phone number it receives are possibly non-constant. On the other hand, the *reboot* method in the *PowerManager* class, for instance, just receives a kernel code for entering special boot modes, which must be part of a pre-defined set of supported flags. This method is thus only a resource method (the data is written into the kernel log), but not an Android Sink. We require this restriction on constant values for methods, which do not introduce any new information into the calling application in the case of sources, or do not directly leak any data across the application boundary in the case of sinks. The values at calls to such methods are of a purely technical kind (e.g., system constants, network pings etc.) and not of interest to typical analysis tools. Note that our definition also excludes some implicit information flows. This is a design choice. For instance, in our approach the vibration state of the phone is not considered a single-bit resource, even though it could theoretically be observed and would then be "shared".

Methods may act as a source or a sink depending on environment settings or parameter values. As a simplified example, imagine a method which first checks whether a SIM card is present and, if so, queries the SIM card for its serial number. This method would only be a source if there is a SIM card in the phone. Since our model of sources and sinks does not contain conditionals, we have to over-approximate and always regard the respective method as a source or sink respectively.

A malicious app can try to access private information not only through calls to the official Android framework API but also through calls to code of pre-installed apps. For instance, the default email application provides a readily-available wrapper around the *getDeviceId()* function. This app is pre-installed on every stock Android phone, which gives a malicious app easy access to the wrapper: the app just instructs the Android class loader to load the respective system APK file and then instantiates the desired class. To cover such cases, our approach does not only analyze the framework API but the pre-installed apps as well. (We use a Samsung Galaxy Nexus with Android 4.2.). In other words, our analysis boundary is between a (potentially malicious) user application and all components pre-installed on the device.

3.4 CLASSIFICATION APPROACH: SUSI

In this section, we explain the details of SuSi, our machine-learning approach to automatically identify sources and sinks corresponding to the definitions given in Section 3.3. We address two classification problems. For a given unclassified Android method, SuSi first decides whether it is a *source*, a *sink*, or *neither*. The second classification problem refines the classification of sources and sinks identified in the first step. All methods previously

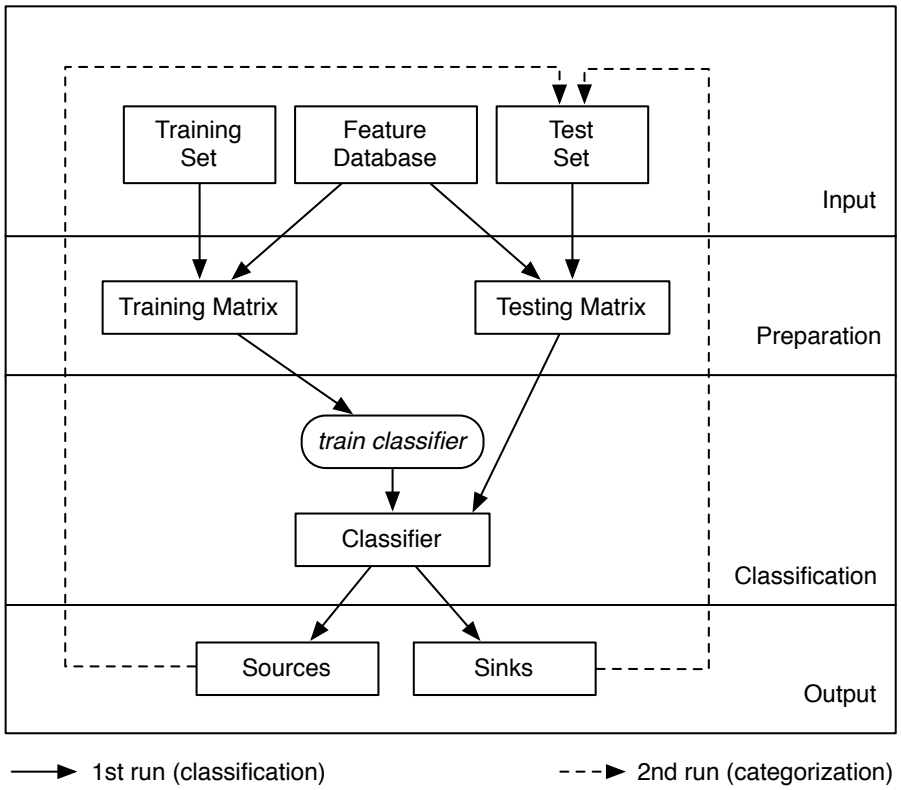

Figure 5: SuSi's machine-learning lifecycle

classified as *neither* are ignored. For an uncategorized source or sink, SuSi determines the most likely semantic category it belongs to. In our design, every method is assigned to exactly one category.

SECTION OUTLINE. Section 3.4.1 presents the general architecture of SuSi, while Section 3.4.2 discusses the features SuSi uses to solve its classification problems. Section 3.4.3 gives more details on one particularly important family of features, which deals with dataflows inside the methods to be classified. In Section 3.4.4 we show how the semantics of the Java programming language can be exploited to artificially generate further annotated training data.

3.4.1 *Design of the Approach*

Figure 5 shows SuSi's overall architecture. It includes four different layers: *input*, *preparation*, *classification*, and *output*. The square elements denote objects, while the round elements represent actions. We run two rounds: One for classifying methods as *sources*, *sinks*, or *neither*, and one for categorizing them. Solid lines denote the dataflow within SuSi. The two dashed lines denote the initialization of the second round. The general process is the same for both rounds. For the categorization, SuSi just takes the outputs of the classification as test data inputs. More precisely, SuSi categorizes separately those methods it has previously identified as sources or sinks and disregards those it classified as neither.

SuSi starts with the input data for the first classification problem, i.e., for identifying sources and sinks. This data consists of the Android API methods to analyze. These meth-

ods can be separated into a set of training data (hand-annotated training examples) and a set of test data for which we do not know whether a method is a source, sink or neither. The set of training data is much smaller than the set of unknown test data, in our case only roughly 0.7% for the classification and about 0.4% for the categorization. Beside the API methods we need a database of features, both for the classification and categorization. The features are different for classification and categorization. See Section 3.4.2 for details.

As described in in Section 3.2, a supervised learning approach requires two matrices. The first one is built by evaluating the features on the set of hand-annotated training data, the second one by applying the same feature set as well to the test data yet to be classified (*preparation* step). SuSi then uses the first matrix to train the classifier (*classification* step), which afterwards decides on the records in the test matrix (*output* step).

While there are a few methods in the Android library that are both sources and sinks (such as some of the *transceive* methods of the NFC implementation), their scarcity stops us from establishing a fourth category "*both*", even though in theory such a category might sound sensible. Classifying a sufficient amount of training data for a machine-learning approach would be equal to classifying almost all transceiver methods. Respectively, we treat such methods as either sources or sinks. This decision affects both the training data and the classifier's results.

In a second step, SuSi *categorizes* the sources and sinks set. In this step, SuSi separately considers the sources and sinks determined in the first step as new test sets (dashed arrows). Note that methods classified as *neither* are ignored at this point. SuSi also requires new training data for the second classification problem. To provide such data, we hand-annotated a subset of the Android sources and sinks with semantic categories related to the mobile domain. We furthermore chose different kinds of features for the feature database as explained in Section 3.4.2. We chose 12 different kinds of source-categories that we identified as being sufficiently meaningful for the different Android API methods: *account*, *bluetooth*, *browser*, *calendar*, *contact*, *database*, *file*, *network*, *nfc*, *settings*, *sync*, and *unique-identifier*. For the sinks, we defined 15 different kinds of categories: *account*, *audio*, *browser*, *calendar*, *contact*, *file*, *log*, *network*, *nfc*, *phone-connection*, *phone-state*, *sms/mms*, *sync*, *system*, and *voip*. For the purpose of compiling our training data, if a method is not relevant or does not fit in any of the identified categories, it is annotated as belonging to the special *no-category* class. If one wants to add a new category, one simply has to create new features for the feature database and randomly annotate the corresponding API methods. Our approach then automatically uses the new feature for the generation of the categorized sources and/or sinks. The subsequent steps as shown in Figure 5 are equal to the ones for the classification. The final output consists of two files, one for the categorized sources and one for the categorized sinks.

Note that some of these categories refer to data being managed by applications, not the operating system itself. One example are contacts: The system provides a data interface to make sure that there is a uniform way of obtaining contacts for all applications that require them, e.g., travel planners, or calendars sending invitations. Additionally, Android contains system applications providing default implementations of these interfaces, so there are methods, which are available on every Android phone and which can be called in order to obtain private data. Therefore, we include categories for such methods, despite them not being part of the operating system as such.

Since we have different categories for sources and sinks, their categorization comprises two distinct classification problems: one for sources and one for sinks. Though they share

the same feature set (see Section 3.4.2), both are solved independently of each other. Thus, quite naturally, the resulting correlations might differ significantly, as some features might be more relevant to distinguish different kinds of sources than different kinds of sinks, and vice versa.

3.4.2 *Feature Database*

We used a set of 144 syntactic and semantic features for classifying the Android methods. A single feature alone does not usually give enough information to decide whether a given Android method is a source, a sink or neither. However, all features in combination can be used to train a highly precise classifier. The same holds for the second classification problem in which we need to find categories for our sources and sinks.

One main reason for why these features work is that many developers of the Android framework do in fact follow a certain regular coding style, or duplicate parts of one method's implementation when implementing another. These social aspects of software development lead to a certain degree of regularity and redundancy in the code base, which a machine-learning approach such as ours can discover and take advantage of.

Though we have a large number of distinct features, most of them are instances of the same parameterized class. For example, the "method name starts with" feature class has instances "method name starts with *get*", "method name starts with *put*", and so on. For identifying sources and sinks, SuSi uses the following classes of features:

- **Method Name:** The method name contains or starts with a specific string, e.g., "get", which can be an indicator for a source.
- **Method has Parameters:** The method has at least one parameter. Sinks usually have parameters, while sources might not.
- **Return Value Type:** The method's return value is of a specific type. A returned cursor, for instance, hints at a source, while a method with a void return value is rarely ever a source.
- **Parameter Type:** The method receives a parameter of a specific type. This can either be a concrete type or all types from a specific package. For instance, a parameter of type *java.io.** hints at a source or a sink.
- **Parameter is an Interface:** The method receives a parameter of an interface type. This is often the case with methods that register callbacks. Note that such methods are neither sources nor sinks according to our definition, since they do not perform any actual operation on the data itself.
- **Method Modifiers:** The method is static/native/etc. Static methods are usually neither sources nor sinks, with some exceptions. Additionally, sources and sinks are usually public.
- **Class Modifiers:** The method is declared in a protected/abstract etc. class. Methods in protected classes are usually neither sources nor sinks.
- **Class Name:** The method is declared in a class whose name contains a specific string, e.g., *Manager*.

- **Dataflow to Return:** The method invokes another method starting with a specific string (e.g., *read* in the case of a source). The result of this call flows into the original method's return value. This hints at a source.
- **Dataflow to Sink:** One of the method's parameter flows into a call to some other method starting with a specific string, e.g., *update*, which would suggest a sink.
- **Dataflow to Abstract Sink:** One of the method's parameter flows into a call to an abstract method. This is a hint for sink as many command interfaces on the hardware abstraction layers are built on top of abstract classes.
- **Required Permission:** Invoking the method requires a specific permission. There is one such feature for every permission declared in the Android API. We were only able to use this feature on the approximately 12,600 methods for which we had permission annotations from the PScout [Au+12] list.

Some features, in particular "Method Name", might sound naive at first, but it turns out that such syntactic features are among the ones that correlate the strongest with sources and sinks. Of course, their effect is only positive in combination with other features; one could not, for instance, detect sources by *only* looking at prefixes of method names.

All our features can assume one of three values: "True" means that the feature applies, i.e., a method does indeed start with a specific string. "False" means that the feature does not apply, i.e., the method name does not have the respective prefix. "Not Supported" means that the feature cannot be decided for this specific method. The latter can happen if, for example, the feature needs to inspect the method body, but no implementation is available in the current Android version's platform JAR file.

The details of our dataflow features are explained in Section 3.4.3. SuSi's features for *categorizing* sources and sinks can be grouped as follows:

- **Class Name:** The method is declared in a class whose name contains a specific string, e.g., *Contacts*.
- **Method Invocation:** The method directly invokes another method whose fully-qualified name starts with a specific string, e.g., *com.android.internal.telephony* for Android's internal phone classes. This feature does not consider the transitive closure of calls starting at the current method.
- **Body Contents:** The method body contains a reference to an object of a specific type, e.g., *android.telephony .SmsManager* for the *SMS_MMS* category.
- **Parameter Type:** The method receives a parameter of a specific type (similar feature as for the classification problem with different instances).
- **Return Value Type:** The method's return value is of a specific type, e.g., *android.location.Country* for regional data.

Note that we do not use permission-based features for the categorization, since many methods require permissions for internal functionality not directly related to their respective category. For instance, a backup method requests many permissions, but does not necessarily give out all of the data it accesses using these permissions if it only creates

an internal save point that can be restored later. The permission list alone thus does not directly relate to the method's category.

It becomes apparent that semantic features are much more suitable for identifying sources and sinks than for categorizing them. On the source-code level, Android's sources and sinks share common patterns, which can be exploited by our dataflow feature. For finding categories, however, there seems to be no such technical distinction and SuSi must rather rely on syntactical features such as class and method names.

3.4.3 *Dataflow Features*

As we found through empirical evaluation, considering a method's signature and the syntax of its method body alone is insufficient to reliably detect sources and sinks. With such features alone we were unable to obtain a precision or recall higher than about 60%. It greatly helps to take the dataflows inside the method into consideration as well. Recall from our definitions in Section 3.3 that sources must read from and sinks must write to resources.

To analyze dataflows, we originally experimented with a highly precise (context-, flow- and object-sensitive) dataflow analysis based on Soot [Lam+11], but found out that this did not easily scale to the approximately 110,000 methods of the Android SDK. Computing precise call graphs and alias information simply took too long to be practical. We thus changed to a much more coarse-grained intra-procedural approximation (also based on Soot[1]) which runs much faster whilst remaining sufficiently precise for the requirements of our classification. Keep in mind that the result of the dataflow analysis is only used as one feature out of many. Thus, it suffices if the analysis is somewhat precise, i.e., produces correct results with just a high likelihood.

Our dataflow features are all based on taint tracking inside the Android API method m to be classified. Depending on the concrete feature, we support the following analysis modes:

- Treat all parameters of m as sources and calls to methods starting with a specific string as sinks. This can hint at m being a sink.
- Treat all parameters of m as sources and calls to abstract methods as sinks. This can hint at m being a sink.
- Treat calls to specific methods as sources (e.g., ones that start with "read", "get", etc.) and the return value of m as the only sink. This can hint at m being a source. Optionally, parameter objects can also be treated as sinks.

Based on this initialization, we then run a fixed-point iteration with the following rules:

- If the right-hand side of an assignment is tainted, the left-hand side is also tainted.
- If at least one parameter of a well-known *transformer* method is tainted, its result value is tainted as well.
- If at least one parameter of a well-known *writer* method is tainted, the object on which it is invoked is tainted as well.

1 We took the *android.jar* built from the OS and the system applications on a real phone (Galaxy Nexus running Android 4.2) as input for Soot.

- If a method is invoked on a tainted object, its return value is tainted as well.
- If a tainted value is written into a field, the whole base object becomes tainted. For arrays, the whole array becomes tainted respectively.

When the first source-to-sink connection is found, the fixed-point iteration is aborted and the dataflow feature returns "True" for the respective method to which it was applied. If the dataflow analysis completes without finding any source-to-sink connections, the feature returns "False".

While such an analysis would be too imprecise for a general-purpose taint analysis, it is very fast and usually reaches its fixed point in less than three iterations over the method body. Since the analysis is intra-procedural, its runtime is roughly bounded by the number of statements in the respective method.

3.4.4 *Implicit Annotations for Virtual Dispatch*

SuSi's implementation is based on Weka, a generic machine-learning tool, which has no knowledge about the language semantics of Java. However, we found that when annotating methods to obtain training data it would be beneficial to propagate method annotations up and down the class hierarchy in cases in which methods are inherited. Such a propagation models the semantics of virtual dispatch in Java. We thus extended SuSi such that if encountering an annotated method *A.foo*, the annotation is implicitly carried over also to *B.foo* in case *B* is a subclass of *A* that does not override *foo* itself, thus inheriting the definition in *A*. Similarly, if *B.foo* were annotated, but not *A.foo*, we would copy the annotation in the other direction.

For our subset of 12,600 methods with permission annotations taken from the PScout list [Au+12], SuSi was able to automatically create implicit annotations for 305 methods. After loading the remaining methods of the Android API to get our full list of 110,000 methods, SuSi was able to automatically annotate another 14 methods.

3.5 EVALUATION

Our evaluation considers the following research questions:

RQ1 Can SuSi be used to effectively find sources and sinks with high accuracy?

RQ2 Can SuSi be used to categorize the found sources and sinks with high accuracy?

RQ3 Which kind of sources and sinks are used in malware apps?

RQ4 How do the sources and sinks change during different Android versions? Can SuSi be used to identify sources and sinks in new, previously unseen Android versions?

RQ5 How complete are the lists of sources and sinks distributed with existing Android analysis tools and how do they relate to SuSi's outputs?

The following sections address these questions in order.

3.5.1 *RQ1: Sources and Sinks*

To assess the precision and recall of SuSi on our training data, we applied a ten fold cross validation and report the results in Section 3.5.1.1. Since the test data used for the cross validation is picked randomly, the results of the cross validation usually carry over to the complete classification performance on unknown training sets if the test set was sufficiently representative. To confirm that this actually holds, we manually evaluated the source and sink lists SuSi generated for the Google Mirror and Google Cast APIs and report the results in Section 3.5.1.2. The Google Cast API is used for the communication between an Android-based smartphone and Google's Chromecast device [Chr]. The Google Mirror API links an Android device to Google Glass [Goo]. We chose these two APIs to show that SuSi is actually able to efficiently handle even previously unseen Android or Java APIs. Note that neither API is included in the base Android system. Secondly, both APIs include methods that handle personal data, such as location or network information. To the best of our knowledge no taint analysis tool has considered these APIs yet. Thirdly, the APIs are of manageable size, making a complete manual validation of SuSi's results practical.

3.5.1.1 *Cross Validation*

We envision SuSi to be used as an automated approach in which experts like us hand-annotate parts of the Android API and then use SuSi to automatically extrapolate these annotations to larger parts of the API. Of course, such an approach only makes sense if the extrapolation is meaningful, which is equivalent to delivering a high precision and recall. Measuring precision and recall is hard in this setting, as one has no gold standard to work with: there is no correctly pre-annotated Android API with which one could compare SuSi's results. Thus, as a best-effort solution we hand-annotated a subset of the Android API ourselves (details below) and then used these methods both as training and test data in a ten-fold cross validation [Koh95], which is the standard approach for evaluating machine-learning techniques. It works by randomly dividing all training data into 10 equally sized buckets, training the classifier on 9 of them, and then classifying the remaining bucket. The process is repeated 10 times, omitting another bucket from training each time. In the end, SuSi reports the average precision and recall. For each class c, precision is the fraction of correctly classified elements in c within all elements that *were* assigned to c. If precision is low it means that c was assigned many incorrect elements. Recall is defined as fraction of correctly classified elements in c within all elements that *should have been* assigned to c. If recall is low it means that c misses many elements.

Table 2 shows the results of this ten-fold cross validation over our training set of 779 methods randomly picked from the PScout subset [Au+12] of about 12,600 methods. The training set contains 13% *source*-, 22% *sink*- and 65% *neither*-annotations. We started with this subset as it provided mappings between methods and required permissions and thus enabled us to also use Android permissions as features for our classifier. The averages we report in our tables are taken from Weka's output. They are weighted with the number of examples in the respective class. Also note that, since our training set is randomly picked, the precision and recall should carry over to the entire Android API with high probability.

Our final results for the source/sink classification had to be computed without any permission features, though, since we do not have permission associations for the complete

Category	Recall [%]	Precision [%]
Sources	92.3	89.7
Sinks	82.2	87.2
Neither	94.8	93.7
Weighted Average	91.9	91.9

Table 2: Source/sink cross validation PScout

Android API[2]. For assessing the impact of the permission feature, we ran the PScout subset again with the permission feature disabled, yielding the results shown in Table 3. Interestingly, the average precision and recall are almost the same with the permission feature and without. The impact of the permission feature is apparently low enough for not having to worry about the lack of permission information when analyzing the complete Android 4.2 API. Conversely, the results also indicate that permissions alone are not a good indicator for identifying sources or sinks.

Category	Recall [%]	Precision [%]
Sources	90.5	91.3
Sinks	86.0	88.8
Neither	95.2	94.4
Weighted Average	92.8	92.8

Table 3: Source/sink cross validation PScout without permission feature

We evaluated SuSi on an extended test set obtained using the implicit-annotation technique explained in Section 3.4.4. With this technique, classifications for a method are copied to all other methods that would lead to the same code being executed according to the semantics of virtual method dispatch in Java. SuSi again shows an average recall and precision of more than 92% (see Table 4). The results are not exactly equal because some of our features consider not just a method's definition but also its container, e.g., the name of the class the method resides in. The fact that SuSi obtains similar results despite these differences is a good indicator of inherent consistency in the results as it shows that semantically equal methods (i.e., ones that have not been overwritten and are thus exposed as-is) are also recognized equally.

Category	Recall [%]	Precision [%]
Sources	89.6	88.0
Sinks	84.7	90.8
Neither	95.2	93.6
Weighted Average	92.3	92.3

Table 4: Source/sink cross validation with implicit annotations

The classifier takes about 26 minutes to classify the complete Android 4.2 API on a MacBook Pro computer running MacOS X version 10.7.4 on a 2.5 GHz Intel Core i5 processor and 8 GB of memory.

2 The available permission lists including PScout are incomplete since they exclude permissions enforced through calls to native code.

As explained in Section 3.2, we experimented with various classification algorithms, and found that SMO performed best. In Table 5, we compare the weighted average precision for SMO, J48, and Naive Bayes, the most well-known representatives of their respective families of classifiers (margin, rule-based and stochastic classifier, respectively). The results were computed on the extended training set obtained through the implicit-annotation technique. The permission feature was not used.

Classifier	**Avg. Recall**			**Avg. Precision**		
	Class. [%]	Source Cat. [%]	Sink Cat. [%]	Class. [%]	Source Cat. [%]	Sink Cat. [%]
Margin (SMO)	92.3	88.8	88.4	92.3	89.7	90.4
Rule-Based (J48)	89.5	81.0	80.2	89.4	81.6	77.4
Probabilistic (Naive Bayes)	86.9	61.5	46.6	87.1	61.7	36.1

Table 5: Source/sink classifier comparison

3.5.1.2 *Validating* SuSi*'s Source/Sink Output*

The output of SuSi's first phase is a list of sources and a separate list of sinks. In this section we verify that the precision and recall of the cross validation in Section 3.5.1.1 is representative for SuSi's actual output. Since manually verifying the outputs for the complete Android API is infeasible, we concentrate on two APIs: The Google Cast API and the Google Mirror API.

Our manual validation of the Google Cast API results in a precision of 96% and a recall of 99% for the sources and a precision of 100% and recall of 88% for the sinks. The somewhat lower recall for the sinks is due the fact this API has only 18 sinks, out of which 16 were detected. The Google Mirror API yields a precision of 100% and a recall of 97% for the sources and a precision of 100% and recall of 94% for the sinks. In result it seems that one can be rather optimistic: at least for these APIs the precision and recall are even higher than the ones obtained through cross validation (cf. Section 3.5.1.1).

3.5.2 *RQ2: Categories for Sources and Sinks*

For evaluating the categorization of sources and sinks, we used similar techniques like the ones used for assessing the identification of sources and sinks in Section 3.5.1. However, recall that only methods identified as sources or sinks in the first step get categorized by SuSi.

3.5.2.1 *Cross Validation*

We used ten-fold cross validation on our training data to assess the quality of our categorization. For this task, we do not use the permission feature, but do apply the implicit annotation technique from Section 3.4.4. Table 6 shows the cross validation results for categorizing the sources, while Table 7 contains those for the sinks.

Category	Recall [%]	Precision [%]
ACCOUNT	100.0	100.0
BLUETOOTH	83.3	100.0
BROWSER	83.0	100.0
CALENDAR	100.0	100.0
CONTACT	95.0	100.0
DATABASE	50.0	100.0
FILE	75.0	100.0
NETWORK	83.3	83.3
NFC	100.0	100.0
SETTINGS	75.0	85.7
SYNC	100.0	100.0
UNIQUE_IDENTIFIER	88.9	100.0
NO_CATEGORY	95.7	62.9
Weighted Average	88.7	89.6

Table 6: Source category cross validation

While SuSi achieves a very high precision and recall for most of the categories, the results for a few categories (e.g., Bluetooth) are considerably worse. These categories are rather small, i.e., randomly picking training methods from the overall set of 110,000 Android 4.2 API methods yields only few entries belonging to such categories. Respectively, there is not much material to train the classifier on. Annotating more data (recall that we only have category annotations for 0.4% of all methods) would certainly improve the situation.

Categories can be ambiguous in some cases. A method to set the MSIDN (the phone number to be sent out when placing a call) could for instance be seen as a system setting (category SETTINGS), but could also be considered a UNIQUE_ID. In such cases, we checked the classifier's result and updated our training data if a misclassification was due to semantic ambiguity, i.e., the result would be right in both categories. Categories that ended up empty or almost empty due to such shifts were removed.

Categorizing the sources took about 6 minutes on our test computer. The sinks were classified in about 3 minutes.

3.5.2.2 *Validating SuSi's Categorized Source/Sink Output*

Manually evaluating the categorized sources and sinks for the Google Cast and Google Mirror APIs shows a precision and recall of almost 100% . The precision and recall for the Google Cast API are 100% for both sources and sinks. For sources in the Google Mirror API the precision is 98% and the recall is 100%. For sinks, both precision and recall are 100%. This shows that the results from Section 3.5.1.2 also carry over to the categorization.

3.5.3 *RQ3: Sources and Sinks in Malware Apps*

It is an important question to ask whether existing malware apps already use sources and/or sinks discovered by SuSi but not currently recognized by state-of-the-art program-

Category	Recall [%]	Precision [%]
ACCOUNT	85.7	100.0
AUDIO	100.0	100.0
BROWSER	50.0	100.0
CALENDAR	100.0	100.0
CONTACT	91.7	100.0
FILE	60.0	100.0
LOG	100.0	71.4
NETWORK	72.7	88.9
NFC	100.0	100.0
PHONE_CONNECTION	75.0	85.7
PHONE_STATE	100.0	100.0
SMS_MMS	96.3	100.0
SYNC	80.0	100.0
SYSTEM	80.6	89.3
VOIP	66.7	100.0
NO_CATEGORY	97.1	70.2
Weighted Average	85.7	88.0

Table 7: Sink category cross validation

analysis tools. To address this question, we selected about 11,000 malware apps[3] from Virus Share [Vir] and analyzed which kinds of sources and sinks these malware samples use. Unsurprisingly, as already found by different researchers [Enc+11b; ZJ12; Gra+12] current malware is leaking privacy information such as location information or the address book.

Interestingly, however, these samples do not only use the standard source and sink methods commonly known to literature, but also such ones not detected by popular program analysis tools (see Section 3.5.5). In total, the samples revealed usage of more than 900 distinct source methods, all of which can be used to obtain privacy-sensitive information. Furthermore, the samples leak data through more than 500 distinct sink methods. The `getLac()` and `getCid()` methods used in our motivating example (see Section 3.1) are two of the most commonly used methods in the LOCATION_INFORMATION category. This is partly related to the fact that both are called in the Google Maps Geolocation API [Geo], which is used in the respective malware samples. Another example is the `getMacAddress()` method in the `WifiInfo` class that SuSi categorizes as NETWORK_INFORMATION. This method is among the most often called methods in this category and is not treated as a source by many tools either. By manual analysis of different malware samples, we found that these source methods are not just called, but their privacy-sensitive return values are indeed leaked to a remote web server.

Since approaches such as LeakMiner [YY12] create their source and sink lists from a permission map, we also analyzed whether malware samples exploit source methods that do not need a permission. Examples of such methods are `getSimOperatorName()` in the `TelephonyManager` class (returns the service provider name), `getCountry()` in the `Locale` class, and `getSimCountryIso` in the `TelephonyManager` class (both return the country code), all of which are correctly classified by SuSi. By manually analyzing the malware samples,

3 Please note that this evaluation was conducted in 2013.

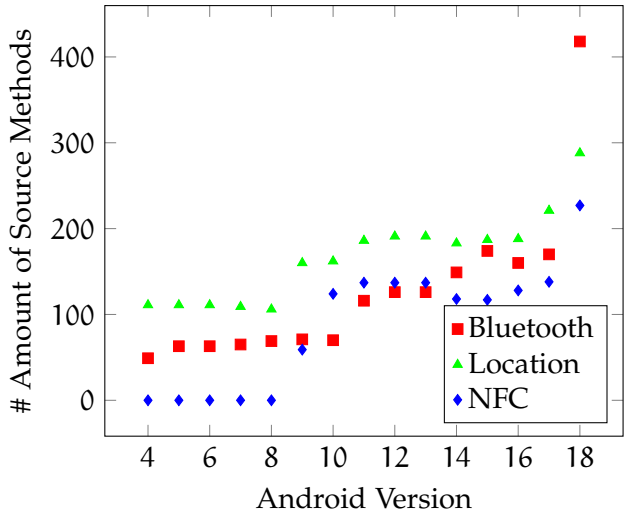

Figure 6: Amount of source methods for bluetooth, location and NFC information in different Android versions

we found that these methods are used frequently and that this data is actually leaked to web servers. This confirms that approaches, which solely rely on the permission map for inferring sources and sinks, miss data leaks in real-world malware samples.

SuSi's categorized output of sources and sinks for Android 4.2 (see Section 3.5.1.1) includes a lot of methods that return privacy-sensitive information, such as the IMEI. SuSi found that there is not only one way of accessing such information (e.g., via `getDeviceID` for the IMEI). Instead, there are plenty of wrapper methods in internal Android classes or pre-installed apps that return the same value. One example would be the internal `GSMPhone` class or the pre-installed email-application which contains a `getDeviceId()` method for returning the IMEI. These methods can only be called using explicit class loading and reflection, but still work on an unmodified stock Android phone. We analyzed the malware samples for this obfuscation technique but found no sample that actually tries to obtain personal data through such methods. Furthermore, we did also not find methods for sinks that are not so well known as shown in the motivating example (cf. Section 3.1). However, we expect such advanced techniques to become more prevalent when security tools evolve, for instance by incorporating the results of this paper, and thus more effectively detect the easier cases.

3.5.4 *RQ4: Changes during Android Versions*

To assess how well SuSi can deal with previously unseen versions of the Android operating system, we compared the categorized source and sink lists generated for a selection of different Android releases[3]. Figure 6 shows the number of sources found for API versions 4 (Android 1.6) to 18 (Android 4.3). We here focus on the *bluetooth*, *location*, and *NFC* categories, as they nicely demonstrate how Android was extended over the various versions. One can deduce from Figure 6 that new sources are introduced with every version. This is yet another motivation to use a tool-supported approach like SuSi's to discover sources and sinks.

The distribution of the number of source methods for location information shows three different jumps, namely between versions 8 and 9, between 16 and 17, and between 17

and 18. This is due to major changes in the Android location APIs [Chaa; Chab; Chac]. The same holds for the jumps in the number of bluetooth information sources between versions 17 and 18, where new source-bearing classes where added to the `android.bluetooth` API. One also clearly sees that NFC was added to Android in API version 9 [Chaa]. There are some cases in which the number of sources is decreased from one version to the next, e.g. between versions 4, 5, 6, 7 and 8 for location. This is related to minor changes in the API. The cross validation results on the different Android versions were effectively the same as reported for version 4.2 in Sections 3.5.1.2 and 3.5.2.

Our results show that SuSi detects the changes in different API versions very well. It reliably finds new sources and sinks that were added to the Android platform and thus provides a much higher level of coverage than available lists assembled by hand. Note that for completely new, previously unanticipated APIs that should yield a new category, SuSi obviously cannot anticipate this category either. In such cases one can easily open a new category, though, by annotating by hand a few examples that fall into this category. This is exactly how we formed categories in SuSi's training set.

3.5.5 *RQ5: Existing Lists of Sources & Sinks*

In this section we assess to what extent current static [MS12; Kim+12; Lu+12; Enc+11b; FCF09; Gib+12; YY12; For; Appe] and dynamic [XSA12; Enc+10] code analysis approaches could benefit from our categorized sources/sinks list[3]. As our results show, SuSi finds all the sources and sinks these previous approaches mention, plus many others which the community was previously unaware of, including some of which are actually being used by malware. Most of the code analysis tools were not publicly available, precluding one from directly comparing their source and sink lists to SuSi's [MS12; Lu+12; Kim+12; Gib+12; XSA12; YY12]. For those approaches we thus estimated the lists from their research papers.

Mann et al. [MS12] mention a few concrete source and sink methods. This hand-picked list is only a fraction of the one produced by SuSi. The taint-tracking tool CHEX [Lu+12] uses a list of 180 semi-automatically collected sources and sinks. Unfortunately, this list is not publicly available and the paper does not explain how the semi-automatic approach works. The authors do mention that their list is based on the Android permission map by Porter Felt et al. [Fel+11] but also argue that this list is insufficient. LeakMiner [YY12] uses the Android permission map to identify sources and sinks. From this map it filters out all methods an application is not allowed to use. However, this leaves open how the tool actually identifies the *relevant* sources and sinks in the remaining method set. Furthermore, if all methods not requiring a permission are filtered, some sensitive data might be overlooked as we have shown. ScanDal [Kim+12] and AndroidLeaks [Gib+12] do not provide concrete lists of source and sink methods. The publications only provide categories (e.g., location information, phone identifier, Internet, etc.), which are also covered by our automatic categorization. Aurasium [XSA12] shifts the problem of identifying sources and sinks by intercepting calls at the system level, i.e., between the native Android libraries and the standard Linux system libraries. While this reduces the number of methods to consider, it makes it harder to reconstruct higher-level semantics, and is failure-prone in case of Android version upgrades. Due to this design, the sources and sinks considered by Aurasium are incomparable to SuSi's results.

Three different taint-analysis approaches were publicly available to us: The dynamic taint analysis tool TaintDroid [Enc+10], an approach based on DeD by Enck et al. [Enc+11b], and

SCanDroid [FCF09]. TaintDroid does not specify the high-level API calls as sources or sinks. Instead, it uses the smaller set of lower-level internal system methods called by those, an approach somewhat comparable to Aurasium. However, this again raises the problem of reconstructing the higher-level context from lower-level calls. The type of data leaked can thus be imprecise. The DeD approach works by decompiling the Android bytecode into Java bytecode that is then used as input for the commercial Fortify SCA [For] static code analysis suite. Fortify can be configured with rules for defining sources and sinks. Enck et al. created such rules and made them publicly available [Enc]. The list contains about 100 Android sources and 35 Android sinks, all of which are also included in SuSi's source and sink lists. For SCanDroid, we extracted the source and sink specifications from the source code (version of April 2013). The resulting list appears hand-picked and is fully covered by SuSi's output.

For evaluating the completeness of the source and sink lists contained in these three tools, we analyzed the most frequently referenced source and sink methods in the malware samples from Section 3.5.3. Table 8 shows that the three tools treat only a few of the methods as a sources or sinks respectively. To assess TaintDroid, we created a separate app for every source and sink in the table. For a source, the respective data is obtained and then leaked via the network (note that the network connection is treated as a sink by TaintDroid). For the sinks we used the well-known `getLongitude()` method as a source (which is treated as a source by TaintDroid) and also created one app per sink. We ran all of our apps on a phone with Cyanogenmod 10 [Cya] containing TaintDroid for Android 4.1. The results of our evaluation are shown in Table 8.

Table 8 shows that the source and sink lists of the three tools are missing some important methods such as one returning the Wi-Fi MAC-address which enables a phone to be uniquely identified. All three tools also miss the method for obtaining the list of accounts (mail, Exchange, social networks, etc.) registered in the phone.

We also found that TaintDroid over-approximates the list of sources and sinks, leading to over-tracking, for instance by tainting the result value of all methods in the *TelephonyManager* class, including the result of *toString()*, which is just the Java object ID (default implementation inherited from *java.lang.Object*). We thus argue that automatically inferring higher-level API methods as provided by our approach would improve tools like TaintDroid as this would allow one to more easily categorize and differentiate various types of sources and sinks.

In total, the results of our evaluation show that obtaining a complete list of sensitive sources and sinks is difficult and SuSi's automatically generated list of categorized sources and sinks can be used to improve this situation.

We also examined well-known commercial tools for static code analysis such as Fortify SCA [For] by HP and IBM AppScan Source [Appe]. As we found, by default these tools provide lists that are rather incomplete. However, both provide an easy way to integrate new sources and sinks to be considered by the analysis. This shows that these tools shift the problem of defining sources and sinks to the analyst, who still needs to obtain such a list from somewhere. SuSi can help to provide more comprehensive defaults.

3.6 APPLICATION SCENARIOS

Apart from the described dataflow scenario, there are various code analysis scenarios, where a comprehensive list of source and sink APIs is necessary. Sensitive API calls pro-

vide a lot of semantic information about the behavior of an application. Code analysis approaches for security problems take advantage of this fact and include this information into their analysis.

MANUAL REVERSE ENGINEERING. During a manual malware investigation, a malware analyst needs to manually reverse engineer an application for understanding a certain behavior of an application. Examples might be the identification of the malicious channels (server, email account, etc.), the identification of the leaked data or the understanding of the functionality of the malware. In all these cases, a comprehensive list of sensitive API calls is indispensable. If the analyst is not aware of certain sensitive API calls that are part of the malware, it is more time-consuming to successfully finish the investigation. In Chapter 6 we introduce a novel Android reverse engineering framework, which makes use of SuSi's output for a faster manual reverse engineering task.

STATIC ANALYSIS APPROACHES. Static approaches are very useful once the analysis needs a more or less complete view of the application. Analyses range from simple analyses to very complex ones. Easy analyses consist of search-based approaches for detecting certain API calls, e.g., used in machine learning approaches [Arp+14; Gor+14]. More complex ones implement a static slicing approach for detecting implementation flaws in cryptographic APIs for instance [Ege+13] and very complex ones implement complex dataflow analysis approaches [Arz+14b]. More concretely, there are already many different approaches that make use of SuSi. For instance, there are approaches that use SuSi's output list as features input for malware detection [Avd+15; JGM15; Bac+14; Tri+16; Li+15c], other use it for dataflow detection approaches [Li+15a; Kli+14; Arz+14b; TS15; Do+16], privacy leak detection [He14; Sla+16] or vulnerability detection [Tia16]. Other approaches use SuSi's output for code obfuscation reasons [Hof14], permission to API method mapping [Bac+16a], library detection [Liu+15] or for developing fine-grained security policy enforcement approaches [Fra+15b; Bag+16; Yu+15]. However, it is also used for loop detection in Android applications [Fra+15a] and automatically generating trust environments for Android applications [Rub+16]. All of them have in common that they operate on the application's bytecode or more concretely on the usage of certain API calls. A comprehensive list of sensitive API calls positively improves the recall of these approaches.

DYNAMIC ANALYSIS APPROACHES. Similar to the static approaches, also dynamic approaches suffer from an incomplete list of sensitive API calls. In the case of dynamic analysis approaches, different techniques are applied such as *bytecode instrumentation* [ARB13], *hooking techniques* [Xpo] or *manipulations of the AOSP* [Enc+10]. For instance, droidfax [CR16] uses bytecode instrumentation and makes use of SuSi's output for detecting malicious applications.

HYBRID ANALYSIS APPROACHES. In this thesis, we will introduce two novel approaches, HARVESTER in Chapter 4 and FuzzDroid in Chapter 5, which are based on a hybrid code analysis. Both approaches require a complete list of sensitive API calls for a better detection rate of concrete insights of an application. Current research approaches use SuSi's output for detecting code vulnerabilities [Has+15; Li+14] or for security policy enforcement [Ras+14] with hybrid approaches.

```
NmeaListener mylistener = new NmeaListener() {
  public void onNmeaReceived(long arg0, String nmea) {
    if (nmea.startsWith("$GPGLL")) {
      String[] data = nmea.split(",");
      Log.d("Loc", "Longitude: "
        + data[3]} + data[4]
        + ", Latitude: " + data[1] + data[2]);
    }
  }
};
LocationManager lm = (LocationManager) this.getSystemService(LOCATION_SERVICE);
lm.addNmeaListener(mylistener);
// Just to start GPS, no data from this callback is ever used
lm.requestLocationUpdates (LocationManager.GPS_PROVIDER, 0, 0, new LocationListener() { ... });
```

Listing 2: Android location leakage via NMEA data

3.7 LIMITATIONS

SuSi works well when it comes to classifying sources and sinks based on their structural similarity to other sources, respectively sinks. In practice, this seems to work well for sources that return data from method calls and sinks that obtain data through parameters. Android offers other less prevalent sources and sinks, however, which cannot be easily classified through machine-learning. We next give some examples.

Applications can implement callback methods and receive data from the operating system through the parameters of these methods. This is commonly used to, e.g., obtain the location in an Android application. In an attempt to avoid detection, the app could however register the callback with *onNmeaReceived* instead of the well-known *onLocationChanged* method and then parse the raw GPS data (the NMEA records) as shown in Listing 2 to get the same data. This shows that a complete list of callback methods is required for finding all data leaks. Due to our definition of sources, SuSi cannot currently find such callbacks. The number of callback interfaces in the Android operating system, however, is sufficiently small for manual inspection. All callback handlers are defined using a small set of well-known and documented interfaces. Static analyses thus aid their detection by finding methods taking these interfaces as parameters. This approach scales well and does not introduce an unreasonable number of false positives as shown in [Arz+14b].

Android defines layout controls through XML files. In the source code, they can be accessed by passing the respective identifier to the system's *findViewById* function. Depending on the ID that is passed, this function can return, for instance, a reference to a password field or to a button with a constant label. Thus, depending on the ID, the method can or cannot be a source. Since calls to this function are present in almost every Android app, a precise analysis must model the Android resource system. If UI sources are restricted to password fields (the default in FlowDroid [Arz+14b]), the analysis scales well in terms of precision. Regarding every input field as a source, on the other hand, can lead to a substantial number of false positives. A more fine-grained tradeoff might be possible by exploiting knowledge about the app's expected behavior.

3.8 RELATED WORK

Our work was originally inspired by Merlin [Liv+09], a probabilistic approach that uses a potentially incomplete specification of sources, sinks and sanitizers to produce a more complete one. Livshits et al.'s approach is based on a *propagation graph*, a representation of the

inter-procedural dataflow in the program where probabilistic inference rules are applied. Their specifications are based on *string-related* vulnerabilities, such as cross-side-scripting vulnerabilities or sql-injections. SuSi in comparison to Merlin does not need any information about the client program or application. It instead analyzes the Android framework code alone to generate a list of categorized sources and sinks. Furthermore, purely string-based approaches fit a web application scenario, while SuSi focuses on privacy-related aspects of Android where data is usually not of type *string* (e.g., the longitude and latitude information is of type *double*).

Privacy violations through leaks of sensitive data in Android applications are well known in the community. To protect the user's privacy, different kinds of taint-tracking approaches have been proposed, both static [Hof+13; Lu+12; YY12; Enc+11b; FCF09; Kim+12; Gib+12; Bat+11; MS12; ZO12; Chi+11; For; Appe; Li+15a; Cao+15] and dynamic [Enc+10; XSA12; Jeo+12]. As already described in Section 3, such approaches are only as good as the source and sink lists they are configured with. In Section 3.5.5 we have shown that all approaches we have evaluated only consider a few sensitive methods for sources and sinks. With the support of our categorized list of sources and sinks, we argue that all of them could be improved to detect more data leaks that are a security problem for the mobile device user. DroidSafe [Gor+15] is a static information flow approach that claims to manually identify all sensitive sources and sinks. This is a very time-consuming and error-prone approach.

More generic policy enforcement approaches such as AppGuard [Bac+13] also require comprehensive lists of sensitive information sources. AppGuard, for instance, provides the user with the ability to revoke permissions after app-installation time. The implementation inserts additional permission checks into the application (not the framework). This requires the identification of relevant methods at the API level for which such checks are required. Our list of sources and sinks includes many methods that require permissions and access sensitive information (e.g., phone identifier, location information, etc.) but are not considered by AppGuard (evaluated version 1.0.3).

Applying machine-learning for security has already been done for automatic spam detection [Sch03] or anomaly detection in network traffic [Seb+02]). Sarma et al. [Sar+12] and Peng et al. [Pen+12] successfully used various machine-learning approaches to detect malicious Android applications. MAST [Cha+13] is a machine-learning approach based on Multiple Correspondence Analysis (MCA) for automatically identifying malicious applications from various Android markets. The tool aims at ranking apps for inspection by a human security analyst, thereby giving priority to those applications that look suspicious. For classifying sources and sinks, we use SMO instead of MCA since MCA requires a logical ordering of records that is not applicable to our scenario. SuSi instead works on discrete and independent classes.

3.9 SUMMARY AND CONCLUSION

In this chapter, we presented SuSi, a novel automated machine-learning guided approach for identifying sources and sinks in the Android framework and pre-installed apps. This approach answers the first research question in the beginning of this chapter and presents the first major contribution of this thesis.

Additionally, with a similar technique as for the identification of sources and sinks, we have further proposed an approach that is capable of automatically categorizing sources and sinks according to the type of data being processed, for instance to distinguish be-

tween sources providing unique identifiers and sources providing file data. This approach answers the second research question stated in the beginning of this chapter.

Method	Description	TaintDroid	SCanDroid	DeD
android.bluetooth.BluetoothAdapter.getAddress()	Returns the hardware address of the local Bluetooth adapter.	no	no	no
android.net.wifi.WifiInfo.getMacAddress()	Returns the MAC address of the Wi-Fi interface.	no	no	no
java.util.Locale.getCountry()	Returns the country code for the phone's locale.	no	no	no
android.net.wifi.WifiInfo.getSSID()	Returns the SSID of the current 802.11 network.	no	no	no
android.telephony.gsm.GsmCellLocation.getCid()	Returns the GSM cell id.	no	no	no
android.telephony.gsm.GsmCellLocation.getLac()	Returns the GSM location area code.	no	no	no
android.location.Location.getLongitude()	Returns the longitude in degrees.	yes	yes	yes
android.location.Location.getLatitude()	Returns the latitude in degrees.	yes	yes	yes
android.accounts.AccountManager.getAccounts()	Returns all accounts of any type registered on the device as a list.	no	no	no
java.util.Calendar.getTimeZone()	Returns the time zone.	no	no	no
android.telephony.TelephonyManager.getDeviceId()	Returns the unique device ID.	yes	no	yes
android.telephony.TelephonyManager.getSubscriberId()	Returns the unique subscriber ID.	yes	no	yes
android.telephony.TelephonyManager.getLine1Number()	Returns the phone number of the device.	yes	no	yes
android.telephony.TelephonyManager.getSimSerialNumber()	Returns the serial number of the SIM.	yes	no	yes
android.provider.Browser.getAllBookmarks()	Returns a cursor pointing to a list of all the bookmarks.	yes	no	no
android.telephony.SmsManager.sendTextMessage()	Send a text based SMS.	yes	yes	yes
android.util.Log.d()	Sends a debug log message.	no	no	yes
java.net.URL.openConnection()	Returns a URLConnection instance that represents a connection to the remote object referred to by the URL.	yes	no	no

Table 8: Detection rate of most frequently used sources and sinks in malware samples [Vir] with different analysis tools

4

RUNTIME VALUE EXTRACTION

In the previous chapter, we proposed a new approach for identifying Android Source and Android Sink API calls in the AOSP and therefore in Android apps. This gives a first assessment of the application, i.e., what sensitive API methods are implemented. However, there are many situations in which more detailed information is required. For instance, the malware investigation described in the motivating example (see Section 1.1.1) required information about the concrete email-username and email-password for a takedown of the malware. Especially for the Android Sinks, it is most of the time required to know concrete runtime values that get passed into API calls.

Runtime values in benign applications are already hard to extract precisely, but modern malware such as Pincer [sec16], Obad [TKG13] or FakeInstaller [Rui12] creates an even greater challenge by obfuscating runtime values deliberately. The malware stores such values (e.g., reflective call targets, the target telephone numbers of SMS scams, or the addresses of remote C&C servers) in an encrypted format inside the application code, to be decrypted only at runtime.

Statically extracting [Hof+13; Bac+16b] these runtime values is practically impossible for many modern malware families such as Obad. Different well-known limitations for static code analysis approaches hinder the extraction. The usage of reflective method calls is one of them. Many current static analyses either do not handle reflection at all or only support constant target strings [Arz+14b; Li+15b; Oct+13; Oct+16; Li+16a]. Therefore, they always have an incomplete picture of the code's behavior, because their handling of runtime values can never be complete. This results in the fact that they are not able to extract runtime values at arbitrary code locations.

If static analysis fails one might think that maybe dynamic analysis can come to the rescue. Current malware, however, also fools dynamic analyses. This is because many malicious applications nowadays contain so-called *time bombs* or *logic bombs* [Coo+09; Pet+14; VC14a]. Logic bombs cause an app to suppress any malicious activity if the app itself detects that it is executing within an analysis environment [Chy15]. Time bombs cause an app to suppress the malicious behavior in any case for a longer period of time, or until after a reboot of the phone, etc. This also includes botnet malware that only acts in response to a command received from a command-and-control server—a command that dynamic analysis tools will find virtually impossible to guess correctly. Moreover, for all applications, including benign ones, a dynamic analysis can only reason about code paths that the analysis actually executes. However, neither an automatic event-generation or UI-testing tool, nor a human analyst can generally cover all possible execution paths in a finite amount of time, causing most dynamic analyses to be incomplete. Even current approaches [JPM13a; CGO15] do not yet scale very well and can take hours even for medium-sized apps. Equally important, Android applications are heavily interactive. To trigger the malicious behavior, certain user interactions may be required. Dynamic tools need to simulate these interactions, as they can gather information only about code paths that they actually execute. Previous work [RCE13; CNS13; HN11] has shown that even for medium-sized Android apps complete code coverage is often impossible to achieve. Consequently, many runtime

values of interest remain unknown when using purely dynamic tools. This makes it very difficult for automatic classifiers or human analysts to detect malicious behavior.

The fundamental research question that we address in this chapter is:

> How can one automatically extract runtime values at a given code location of modern Android malware applications, even if this application uses common anti-static and anti-dynamic code obfuscation techniques?

In this chapter, we present HARVESTER, a novel approach that seeks to effectively address all of the above problems for current obfuscated malware samples. Even for the most sophisticated malware families such as Obad, Pincer, or FakeInstaller, HARVESTER is able to extract virtually all runtime values of interest within minutes, without any user intervention, and in our experiments with perfect accuracy.

Contributions. In summary, this chapter provides the following original contributions:

- a variation of traditional slicing algorithms fine-tuned to support the hybrid extraction of runtime values in Android applications,
- a dynamic execution system for running the computed code slices and extracting the values of interest without user interaction,
- an evaluation of the approach's feasibility for a mass-analysis on real-world malware applications, and
- three case studies assessing how HARVESTER can improve the coverage of existing off-the-shelf static and dynamic analysis tools.

Chapter Outline. In the beginning of this chapter, we explain the basic principles of the HARVESTER approach on a motivating example (Section 4.1) derived from the main motivating example (Section 1.1.1). Afterwards, we provide some definitions in Section 4.2. The approach is explained and evaluated in Section 4.3, followed by an explanation of different application scenarios in Section 4.4. Then, limitations of the approach (Section 4.5) and related work (Section 4.6) are presented. A conclusion and summary concludes this chapter in Section 4.7.

4.1 MOTIVATION AND CONTRIBUTION

Listing 3 shows a slightly modified version of the data leakage part of the motivating example in Section 1.1.1. The example heavily relies on obfuscation to hide its behavior from both analysis tools and manual investigators. At runtime, instead of calling methods directly, the example takes a string previously encrypted and decrypts it using a lookup table. It then uses reflection to find the class and method that bear the decrypted name and to finally invoke the retrieved method.

Many current malware applications are obfuscated in a similar way, either manually or by using commercial tools such as DexGuard [Tec14]. For a human analyst to understand the runtime behavior of such obfuscated code, she must know the target methods of the reflective calls. In the example, these values are the decoded class name in line 18 and

```
1  class SMSReceiver extends BroadcastReceiver {
2  private native String getAccountName();
3  private native String getAccountPassword();
4
5  void onReceive(SMS sms) {
6    String body = sms.getSMSBody();
7    String accountName = getAccountName();
8    String accountPassword = getAccountPassword();
9
10   //emulator-check
11   if(!isEmulator()) {
12     //stores country information
13     String countryInfo = simCountryIso().equals("US") ? US : INTERN;
14     int cellID = CellLocation.getCid();
15     int lac = CellLocation.getLac();
16     String emailBody = countryInfo + " : " + cellID + " : " + lac;
17     //class: MailSender class
18     String clazzString = decrypt("1234", "ai03_");
19     //method: sendEmail
20     String methodString = decrypt("1234", "fahg29favjvajii");
21     Method method = Class.forName(clazzString).getMethod(methodString, String.class, String.class,
        String.class);
22     //MailSender.sendEmail(emailBody, account-name, account-password)
23     method.invoke(emailBody, accountName, accountPassword);
24   }
25 }
26 }
```

Listing 3: Motivating example HARVESTER: obfuscated code that sends an email under certain conditions

the decoded method name in line 20. To find these values manually, she would have to carefully inspect the decompiled bytecode, find the lookup table, and manually decrypt all strings to detect the malicious behavior. Strings decrypted for one application once cannot usually be reused, as different malware variants use different lookup tables.

Static code analysis approaches such as SAAF [Hof+13] apply techniques like backward slicing in order to extract constant string information. These tools, however, have well-known limitations that make them fail on highly obfuscated applications, e.g., ones with dynamically-computed values as shown in Listing 3. Even those static-analysis tools that model the full string API still have limitations that can easily be exploited by malware developers. For example, one can implement the string-decoding method in a custom library written in native code. To the best of our knowledge, no static analysis tool for Android supports such native code.

The code in the example challenges dynamic analysis approaches as well. The analyses first has to send an SMS to the device for executing the `onReceive()` callback. Then, it has to pass the `isEmulator()` check, which checks whether the applications runs on an emulator or a real device [RKK07; Pet+14; VC14a]. Dynamic analysis environments can never fully hide all of these characteristics [Coo+09] and thus fail on sophisticated malware.

HARVESTER, on the other hand, fully automatically retrieves all relevant runtime values of the example in Listing 3. The security analyst simply specifies the variables for which runtime values should be retrieved. For the example, we assume that the security analyst knows that she is interested in the parameters given to any calls to `MailSender.sendEmail()` (pseudo API call representing the sending of an email) that the application may make. As one can easily see, the code in Listing 3 contains no direct call to this API. Instead, the calls to this API are issued through reflection. But HARVESTER comes pre-configured with a setting that further extracts the parameters to such reflective calls, and inlines calls accordingly, once discovered.

In a first step, HARVESTER would hence attempt to extract parameters to the `forName()` and the `getMethod()` calls (line 21). HARVESTER's static slicer automatically extracts all code computing those values, while *crucially*, however, discarding certain conditional control-flow constructs that do not impact the computed value (concrete details will be provided in Section 4.3.2). In the example, this will discard the environment-detection check at line 11. HARVESTER's dynamic component then runs only the reduced code. Since all environment-detection checks are eliminated, the dynamic analysis immediately executes all those parts of `onReceive()` relevant to the computation of the selected values. At runtime, the analysis discovers the name `MailSender.sendEmail()` of the method called through reflection. In result, it replaces the original reflective method call by a direct call to that very API, and re-iterates the extraction process.

Assuming that the security analyst configured HARVESTER to extract the arguments given to such calls, HARVESTER performs a slicing for `emailBody`, `accountName` and `accountPassword`. It extracts the corresponding slices in the same way as before, the reduced code is executed, and HARVESTER reports the concrete email account credentials and data sent via email.

Note that HARVESTER does not require any manipulations to the underlying Android framework. It works purely on the bytecode level of the target application, through a bytecode-to-bytecode transformation.

4.2 LOGGING POINTS AND VALUES OF INTEREST

The main purpose of HARVESTER is to compute runtime values. Formally, we call these runtime values *values of interest*. To use HARVESTER, a human analyst defines *logging points* for which she wants to extract all values of interest. Both are defined as follows.

Definition 1 *A* logging point $\langle v, s \rangle$ *comprises a variable or field access* v *and a statement* s *such that* v *is part of* s*.*

Definition 2 *A* value of interest *is a concrete runtime value of variable* v *at a logging point* $\langle v, s \rangle$*.*

For instance, if one is interested in runtime values passed to a conditional check `s: if(a.equals(b))` the runtime values of `a` and `b` are both values of interest at this statement `s`, inducing the two logging points $\langle a, s \rangle$ and $\langle b, s \rangle$. Another example would be an API call to the `sendTextMessage` method such as `s: sendTextMessage(targetNumber, arg2, messageText, arg4, arg5)` where $\langle$ `targetNumber, s`$\rangle$ and $\langle$ `messageText, s`$\rangle$ are possible logging points at `s`. Parameters `arg2`, `arg4` and `arg5` can be also defined as logging points, but do not provide security-relevant information. The corresponding runtime values are the values of interest. Examples for values of interest for `targetNumber` would be `'+01234'` and for `messageText` would be `'This is a premium SMS message'`.

To ease the definition of logging points for the human analyst, HARVESTER provides a comprehensive list of pre-defined logging points taken from the output of the previous chapter (see Chapter 3). Susi is a machine-learning approach, which provides a comprehensive list of categorized sensitive API methods. HARVESTER makes use of these sensitive API methods by providing generic categories such as *URL*, *Shell-Command* or *SMS Number* as tool-input parameters. For instance, if one is interested in URLs inside the application, one can run HARVESTER with the *URL* parameter and all API calls that are able to call a

```
1 String clazzString = decrypt("1234", "ai03_");
2 String methodString = decrypt("1234", "fahg29favjvajii");
3 Harvester.report(clazz, method);
4 Method method = Class.forName(clazzString).getMethod(methodString, String.class, String.class,
      String.class);
```

Listing 4: Sliced version of the onReceive() method of Listing 3 (part 1)

URL are automatically defined as logging points. This is the only human interaction that HARVESTER requires.

4.3 GENERIC HARVESTER APPROACH

In the following we will explain the HARVESTER approach first from a high level perspective (Section 4.3.1) and continue with a detailed description of the approach in Section 4.3.2. An evaluation in Section 4.3.3 concludes this section.

4.3.1 *Overall Approach*

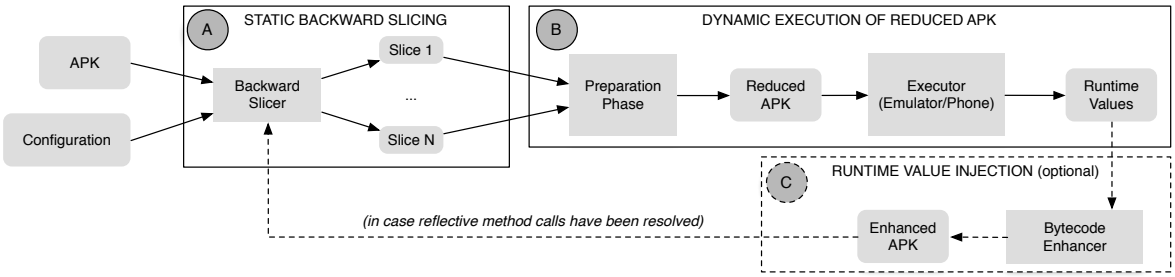

Figure 7: Workflow of HARVESTER

Figure 7 depicts HARVESTER's workflow. To compute values of interest, HARVESTER first reads the APK file and a configuration file defining the logging points. HARVESTER next computes a static backwards slice starting at these code points, as will be further explained in Section 4.3.2.1. This slicing step runs on a desktop computer or compute server. The pre-computed slices are then used to construct a new, reduced APK file, which contains only the code required to compute the values of interest, and an executor activity. The task of the executor activity is to invoke the computed slices and report the computed values of interest.

HARVESTER additionally alters those conditionals whose value depends on the execution environment and on which the slicing criterion, i.e., the value in question, is data-dependent. These conditionals are replaced by simple Boolean variables, allowing HARVESTER to force the simulation of different environments at runtime. Listing 4 shows the output of Harvester when requesting a slice for the parameters of the reflective call in line 4. Line 3 extracts all runtime values that are necessary to replace the reflective call with its API call. As a second step, if the extraction was successful, it adds the original API call into the bytecode. In our example, the `MailSender.sendMail()` will be added to the bytecode. If one is interested in the runtime values for the logging points `emailBody`, `accountName` and `accountPassword`, HARVESTER starts again with its analysis and generates a new slice.

The results can bee seen in Listing 5. This Listing includes the replaced reflective method call and the corresponding slice for the new API call. Please note that Listing 3 represents

```
1 String countryInfo = EXECUTOR_1 ? US : INTERN;
2 int cellID = CellLocation.getCid();
3 int lac = CellLocation.getLac();
4 String emailBody = countryInfo + " : " + cellID + " : " + lac;
5 Harvester.report(emailBody, accountName, accountPassword);
6 MailSender.sendEmail(emailBody, accountName, accountPassword);
```

Listing 5: Sliced version of the onReceive() method of Listing 3 (part 2)

the replacement of the reflective method call in a simplified way; more details will be explained in Section 4.3.2.3. One can see that the emulator check has been removed, as the slicing criterion is reachable only if the branch falls through. The condition in line 11 has been replaced by the global variable `EXECUTOR_1`, making the slice parametric.

This new, reduced APK file is then executed on a stock Android emulator or real phone, as we explain in Section 4.3.2.2. These steps are fully automated and no user interaction is required. In a *forced execution*, HARVESTER explicitly triggers all the different behaviors of the parametric slice (in Listing 5 with both **`true`** and **`false`** for `EXECUTOR_1`) which allows the complete reconstruction of the values of interest, for all concrete environments, decrypting any encrypted values. HARVESTER instruments the reporting mechanism for the values of interest into the slices (see line 5), making changes to the runtime environment (emulator, Android OS) unnecessary. Note that HARVESTER does not need to reconfigure or reset the actual device or emulator on which the slices are executed which is novel in comparison to other approaches that are based on symbolic or concolic execution [Xu+14; MKK07].

4.3.2 *Detailed Solution Architecture*

Next we provide more details about the main components of HARVESTER namely the *static backward slicing* process, the *dynamic execution of the reduced APK* and the *injection of runtime values into the APK*, as shown in Figure 7. In order to explain these steps, we will first assume that our approach is able to replace the reflective method call in line 23 of Listing 3 into its original method call `MailSender.sendEmail()` (see line 18 Listing 6). The explanation in this section will use Listing 6 as a working example and will explain the steps that are required to replace the reflective method call in the end of this section (see Section 4.3.2.3).

4.3.2.1 *Static Backward Slicing*

Part A comprises the *static analysis* phase. In traditional slicing as defined by Weiser [Wei81], a program slice S is an *executable program* that is obtained from a program P by removing statements such that S replicates the behavior of P [Tip94] with respect to the so-called *slicing criterion*—a value of interest selected by the user. We use Figure 8 to explain the effect of traditional slicing on our initial example from Listing 6. Assume that we want to slice this program such that the parameters `emailBody`, `accountName` and `accountPassword` passed to the `sendEmail()` method call (line 18) are our slicing criteria. The reflective call is data-dependent on all four assignments to those three variables. The assignments to `countryInfo` are further control-dependent on the check of the `simCountryIso()`. All of those statements are further control-dependent on the check on `isEmulator()`, the environment check that circumvents the execution of the remaining code on Android emulators. Traditional slicing approaches such as the one by Weiser [Wei81] would include this check

```
1  class SMSReceiver extends BroadcastReceiver {
2  private native String getAccountName();
3  private native String getAccountPassword();
4
5  void onReceive(SMS sms) {
6    String body = sms.getSMSBody();
7    String accountName = getAccountName();
8    String accountPassword = getAccountPassword();
9
10   //emulator-check
11   if(!isEmulator()) {
12     //stores country information
13     String countryInfo = simCountryIso().equals("US") ? US : INTERN;
14     int cellID = CellLocation.getCid();
15     int lac = CellLocation.getLac();
16     String emailBody = countryInfo + " : " + cellID + " : " + lac;
17     //de-obfuscated reflective method call
18     MailSender.sendEmail(emailBody, accountName, accountPassword);
19   }
20 }
21 }
```

Listing 6: De-obfuscated code that sends an email under certain conditions

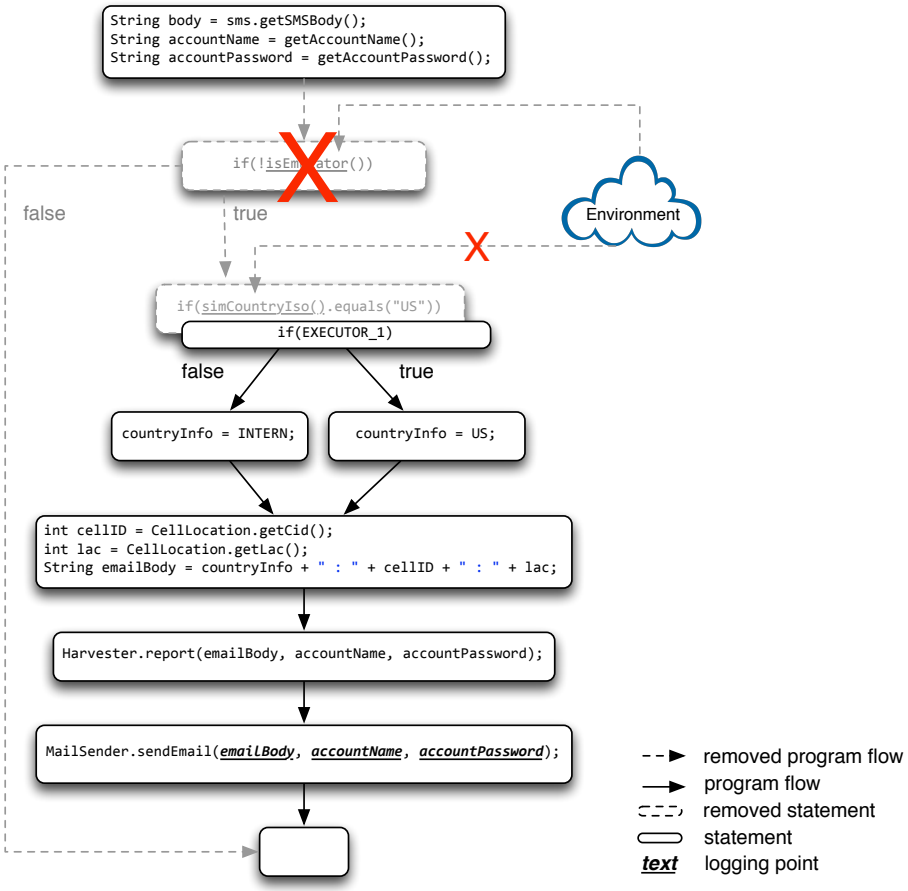

Figure 8: Slice representation of Listing 6

```
1 int valueOfInterest = 120, i = 0;
2
3 label1: if (i < 3) {
4   i++;
5   valueOfInterest++;
6   goto label1;
7 }
8
9 send("" + valueOfInterest, "Hello");
```

Listing 7: HARVESTER's handling of conditions that are not environment dependent

in the slice. Executing the check, however, immediately leads to leaving the method, consequently never triggering the "interesting code" that computes the values relevant to the slicing criterion.

Even if the emulator check were removed, this traditional approach would still not be sufficient as it would still retain the environment-dependent check on `simCountryIso()`. In the specific scenario of malware analysis, the method `simCountryIso()` will return exactly one of several country codes, depending on the configuration of the emulator. But frequently, the malware analyst is interested in inspecting *all possible* runtime values in question. In the example, we would like to cover both possible branches. Without further extensions to the approach this would require a reboot and reconfiguration of the emulator, which is a time consuming and error-prone undertaking. However, while the assignments to `countryInfo` are *control*-dependent on `simCountryIso()` and thus also on the execution environment, there is no *data* dependency. HARVESTER exploits this fact by replacing the conditional referring to `simCountryIso()` by a simple global Boolean flag `EXECUTOR_1`. This flag causes the slice to become *parametric*: the selection of any concrete Boolean values for the generated control variables allows the direct execution of one of the parametric slices. This effectively breaks the dependencies of the app's execution on its execution environment, depicted by the lower red cross in Figure 8.

The same concept also applies to the dependency on `!isEmulator()`. In this case, the code of interest is, however, only executed if this check returns **`true`**. In other cases, the whole computation of the values of interest would be skipped. Therefore, this conditional is replaced by **`true`**, resulting in a removal of the condition as shown at the big red cross.

Note that HARVESTER only parameterizes the slice at those conditionals that are data-dependent on environment values, while all other conditionals remain unchanged. This, for instance, allows HARVESTER to swiftly recover the correct value 123 for the variable `valueOfInterest` in Listing 7, which contains no such reference. Please note that in this snippet we show a **`goto`** operation. HARVESTER works directly on the bytecode level, where all loops are expressed that way. If HARVESTER were to replace all conditionals regardless of whether they are environment-dependent or not, the slice for the example in Listing 7 would compute the incorrect value 120 when choosing **`false`** for the condition `i < 3`. Worse, when choosing **`true`** for the condition, the code would loop infinitely. At this point it is important to note that HARVESTER maintains the assignment to control variables fixed per run, i.e., it can only execute loops never (condition is **`false`**) or infinitely often (condition is **`true`**). In particular, in cases in which a loop condition does depend on an environment value, this may cause one of HARVESTER's dynamic runs to loop indefinitely. HARVESTER simply addresses this problem with a timeout on the overall execution time for every run of a slice. As evident from our experiments, this theoretical shortcoming does not pose a problem in practice. Developers intend computed values

such as reflective call targets or telephone numbers for SMS scam to be independent of the execution environment.

In general, HARVESTER replaces only such conditionals that access values dependent on the execution environment. To be able to determine such conditionals, HARVESTER comes pre-customized with a configuration file listing fields and methods whose return values are known to depend on environment settings. Vidas et al. [VC14b] analyzed different techniques for Android emulator-detection and Maier et al. [MMP14] showed fingerprinting techniques for mobile sandboxes. We use the methods from these papers as a starting point for constructing the required lists. We believe the lists to be complete for current Android versions, but they can easily be extended.

The remainder of this section discusses the most important challenges that arise during backward slicing and how HARVESTER overcomes them.

DATA DEPENDENCIES THROUGH PERSISTENT STORAGE Most applications use API classes such as `SharedPreferences` to persist data. Storage and retrieval can be distributed over the program. For instance, data can be stored into a file during application startup and read again after the reboot of the application—a common workflow also in current Android malware applications [Ras+15b]. A slicing approach that does not model this data dependency between user actions would yield an incorrect slice that attempts to read non-existent data from an uninitialized data store. To handle these cases, HARVESTER resolves *all* calls within the analyzed bytecode that write to persistent storage and prepends them to the slice. This approximation may, however, miss some of the data if the stored value is ambiguous, as only the last value is retained and all earlier values are overwritten. While a better handling might seem desirable, in our experiments the current solution proves sufficient to produce correct values for all logging points.

USER INPUT Further special handling is required for API calls that access environment values such as free-text user input. It is one major contribution of this work to show that within the slices that are frequently of interest to security analysts, such accesses to environment values, are, however, typically restricted to conditionals (see Section 4.3.3). Thus, they are removed by HARVESTER, as the respective expressions are replaced by Boolean control variables. Semantically, this restriction applies because obfuscators and malware authors seek to encode values independently of user input. The target of reflective call, for instance, is assumed to always be the same, regardless of the environment. In some few slices of interest, however, we found user input to be accessed also outside conditionals. In some cases this can simply happen because the slice is less precise than one would like it to be. To allow the execution also of such slices without user interaction, HARVESTER injects code to short-circuit the actual API calls that read out the UI, returning dummy values instead. Dummy values for a string can be `"This is a dummy string"`, which might cause an exception once the slice gets executed. One reason therefore are string operations such as `split("\|")` that are part of the slice. Our experiments show this workaround, albeit somewhat crude, to be highly effective when applied to current malware. However, we improved this situation with our FUZZDROID approach, which is based on a target fuzzing strategy (more details in Chapter 5).

DYNAMIC CODE LOADING AND NATIVE CODE As demonstrated by Figure 8, HARVESTER can also cope with native methods, as long as all logging points are

contained within the APK's Dalvik bytecode. The same applies for dynamic code loading. Two logging point ($\langle \mathit{accountName}, s \rangle$ and $\langle \mathit{accountPassword}, s \rangle$ for statement `s: MailSender.sendEmail(emailBody, accountName, accountPassword)`) at the `MailSender.sendEmail()` method call are computed by a native method. The slicer will declare this function as required and the dynamic execution will evaluate the function just as any other, invoking the same implementation that would also be invoked during normal app execution. If a native method call is part of the slice, the slice will be extended with the `System.loadLibrary()` method call, which is part of the bytecode. Many current malware samples encode important values in native or dynamically loaded code, making this an essential feature [Ras+15b].

CUT-OFFS FOR LARGE PROGRAMS For very large programs it may be infeasible to compute exact slices. HARVESTER therefore supports cut-offs that prevent it from walking further up (into callers) or down (into callees) along the call stack while slicing. After the cut-off, all further callees are retained as they are, without any slicing. All callers exceeding the cut-off are simply disregarded, i.e., HARVESTER, assumes that the slice constructed so far does not depend on any earlier program logic. To avoid uninitialized variables in this case, HARVESTER inserts artificial initialization statements that assign dummy values. As our experiments show, only few such dummy values are required in practice (see Section 4.3.3).

4.3.2.2 *Dynamic Execution of Reduced APK*

Part B in Figure 7 describes the *dynamic analysis* phase. HARVESTER assembles every slice computed during the static slicing phase within a single new method that becomes part of the reduced APK. The executor activity, injected into the same APK file, calls all these methods one after another, directly after the new app has been started, e.g., on an unmodified emulator or a stock Android phone. The executor writes the computed runtime values into an SQLite database on the device's SD card that can then be downloaded and evaluated on a desktop computer. Since the slices are directly executed, regardless of their original position in the application code, HARVESTER requires no user interaction that might otherwise be necessary to reach the code location of the computing statements. If, for instance, the extracted code was originally contained in a button-click handler, it would have required the user or an automated test driver to click that button to be executed. HARVESTER, however, executes the sliced code directly, making these interactions unnecessary. In fact, the reduced app does not contain any GUI (Graphical User Interface) elements from the original app at all. Figure 9 shows how the slice explained in Figure 8 would be executed. Executing this program will cause HARVESTER to report both possible valuations for `emailBody`, along with the values for `accountName` and `accountPassword`.

The reduced app is packaged with the same resources as the original app, such that code that would load encrypted strings, for instance, from external resources, will find those resources also in the reduced APK.

As explained in Section 4.3.2.1, slices are parametric and HARVESTER must explore every possible combination of branches to retrieve all values of interest at a given logging point. For the executor, this means that it must re-run the code slice for all possible combinations of these Boolean values. In the worst case (all conditions in the slice have to be replaced), this leads to 2^n paths where n is the number of conditionals between the introduction of the variable and the position of the logging point. Only conditions inside the

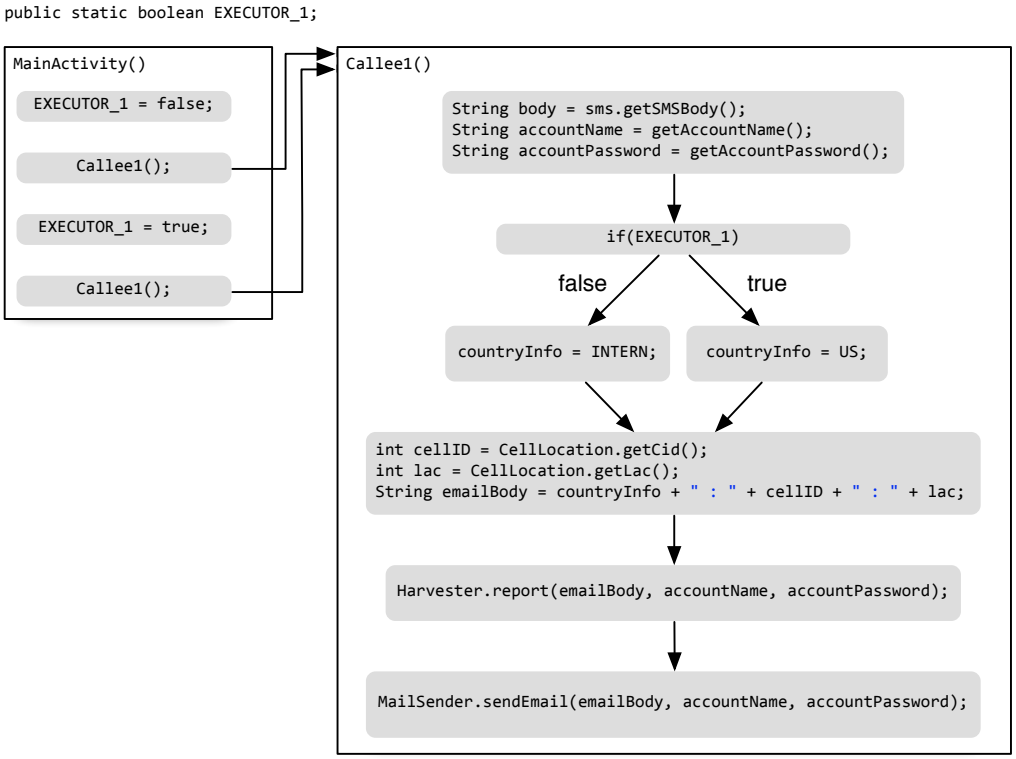

Figure 9: Dynamic execution of reduced APK

slice need to be considered. Thus, in practice, our experiments show n to be very limited ($n = 0.21$ per path on average over all our sample data, see Section 4.3.3). In the few cases in which it is not, our experiments show many of those paths to yield the same or at least very similar values. HARVESTER can thus be configured to sample only a pre-defined number of slice instances at random.

4.3.2.3 *Runtime Value Injection*

Part C in Figure 7 shows an optional step of HARVESTER, *runtime value injection*. This step can be useful to combine HARVESTER with existing off-the-shelf analysis tools, or to handle reflection. Static-analysis approaches require a callgraph to determine potential targets for method invocations. For the large fraction of malware applications that are obfuscated using reflective method calls, such as the example in Listing 3, a construction of the callgraph fails. Some tools do not support reflective calls at all, while frameworks such as DOOP [BS09] implement a static best-effort solution but can still be fooled through string encoding. HARVESTER, however, can aid those off-the-shelf tools by manifesting the runtime values of reflective call targets resolved during the dynamic execution as ordinary method calls in the application's bytecode. This allows existing callgraph construction algorithms to construct a sound callgraph with ease. To embed reflective calls into the program, HARVESTER uses the same approach originally taken in the TamiFlex tool [Bod+11].

Line 23 in our original motivating example in Listing 3 shows such a reflective method call. As a first step, HARVESTER extracts the runtime values for `clazzString` and `methodString` and replaces the `invoke()` method call with the original API call. Listing 8 shows the enhanced code snipped. The replaced API call in line 25 is surrounded with a

```
1  private native String getAccountName();
2  private native String getAccountPassword();
3
4  void onReceive(SMS sms) {
5    String body = sms.getSMSBody();
6    String accountName = getAccountName();
7    String accountPassword = getAccountPassword();
8
9    //emulator-check
10   if(!isEmulator()) {
11     //stores country information
12     String countryInfo = simCountryIso().equals("US") ? US : INTERN;
13     int cellID = CellLocation.getCid();
14     int lac = CellLocation.getLac();
15     String emailBody = countryInfo + " : " + cellID + " : " + lac;
16     //class: MailSender class
17     String clazzString = decrypt("1234", "ai03_");
18     //method: sendEmail
19     String methodString = decrypt("1234", "fahg29favjvajii");
20     Method method = Class.forName(clazzString).getMethod(methodString, String.class, String.class,
        String.class);
21
22     //injected code beginning
23     String methodSig = method.toGenericString();
24     if(methodSig.equals("public void MailSender.sendEmail(java.lang.String, java.lang.String,
        java.lang.String)"))
25       MailSender.sendEmail(emailBody, accountName, accountPassword));
26     else
27     //injected code end
28       method.invoke(emailBody, accountName, accountPassword);
29   }
30 }
```

Listing 8: Replaced reflective method call of Listing 3

check (line 24) whether the extracted method signature is the expected one. Moreover, if HARVESTER would extract more runtime values (more method signatures) for a particular example, it would proceed in the same way and creates an **`if-else`** statement for every API call. To allow for cases in which there are further call targets, which HARVESTER failed to detect dynamically, the old reflective call is nevertheless retained in the fall-through branch (line 28)[1].

Off-the-shelf analysis tools such as CHEX [Lu+12], SCanDroid [FCF09] or FlowDroid [Arz+14b] can then analyze the enriched APK file without requiring special handling for reflection or string operations used to build the target method name. To the best of our knowledge, HARVESTER is the first fully automated approach that performs such a value injection for Android.

It is important to note that this very same mechanism is also what allows HARVESTER itself to extract runtime values from applications whose API calls have been obfuscated through reflection. In such cases, in phase A HARVESTER would first construct a partial callgraph that is incomplete in the sense that it misses edges for reflective calls. It then extracts information about the parameters to those calls and inlines the calls as regular method calls. Finally, it reiterates the process, constructing a new, more complete callgraph, and extracting further data values. This can be iterated up to a pre-defined number of times, or until a fixed point has been reached. This step is shown in Figure 7 by an edge from *Enhanced APK* to *Backward Slicer*.

1 This trick is adopted from the Booster component of the TamiFlex tool [Bod+11] for Java.

The enriched APK files are functionally equivalent to their respective originals and only use normal application-level code. Running them does not require any changes to the operating system or the emulator.

The current prototype supports this technique for reflective method calls only, but it could easily be extended for other obfuscated strings as well. Examples are the target telephone numbers of SMS messages, to aid existing pattern-based malware-detection tools. Injecting the action strings and URIs of Android intents used for inter-component communication is also important. Many static analyses fail if these strings are not constant, as they can no longer map intent senders and receivers. The same applies to class-name strings used with explicit intents. If they are only decrypted at runtime, static analyses have no chance but to conservatively assume all possible recipients, which is highly imprecise. Injecting these strings as constants enables tools such as EPICC [Oct+13] or IC3 [Oct+15] to reconstruct the inter-component callgraph more precisely and correctly identify the dataflows between components and applications, which would otherwise not be possible. In Section 4.4.2, we evaluate how HARVESTER's output can be used to improve the construction of inter-component callgraphs.

4.3.3 *Evaluation of the Generic Approach*

We evaluated HARVESTER extensively on different sets of applications, one to address each of the following four research questions. In total, all sets together, comprise 16,799 apps.

- **RQ1**: What is HARVESTER's precision and recall?
- **RQ2**: How does the recall of HARVESTER relate to that of existing static and dynamic-analysis approaches?
- **RQ3**: How efficient is HARVESTER?
- **RQ4**: Which interesting values does HARVESTER reveal?

In all experiments, the cut-offs were 20 for caller-slicing and 50 for callee-slicing which proved to be a reasonable tradeoff between recall and performance.

RQ1: What is HARVESTER*'s recall and precision?*

We evaluated HARVESTER's recall based on the coverage of logging points. Ideally, HARVESTER should cover every logging point. For the covered logging points we furthermore evaluated the precision and recall of the extracted runtime values. From our initial malware set of 16,799 samples, we took 12 different malware samples from 6 different malware families for an in-depth evaluation as shown in Table 9. These samples were selected since they are representatives of various challenges for HARVESTER. Obad [TKG13], for instance, is one of the most sophisticated malware families today. FakeInstaller and GinMaster are also highly obfuscated. These samples heavily rely on reflection to mask the targets of method calls. Another malware family, Pincer, is known to hinder dynamic analysis through anti-emulation techniques [Pet+14; VC14a]. Ssucl and Dougalek steal various private data items. We deliberately chose 12 complex samples only, since we sought to manually verify the precision and recall of HARVESTER.

	URI	Webview	SMS No.	SMS Text	File	Reflection	Shell Cmd	Sum
FakeInstaller (MD5)								
b702b545d521f129e8efc1631a3abcee	3/3 (0/3)	4/4 (0/4)			6/7 (1/7)	6/6 (0/6)		19/20 (1/20)
dd40531493f53456c3b22ed0bf3e20ef						248/280 (0/280)		248/280 (0/280)
GinMaster (MD5)								
0878b0bb41710324f7c0650daf6b0c93	4/12 (4/12)	0/2 (1/2)			10/14 (2/14)	0/3 (1/3)		14/31 (8/31)
ebe49b1b92a3b44eb159d15ca1f25c70	7/9 (2/9)	1/1 (0/1)			25/30 (3/30)			33/40 (5/40)
Obad (MD5)								
e1064bfd836e4c895b569b2de4700284						185/185 (0/185)		185/185 (0/185)
dd1a3ff43330165298db703f7f0626ce						157/161 (2/161)		157/161 (2/161)
Pincer (MD5)								
b2b7d5999dce0559d13ab06d30c2c6ec	2/2 (0/2)		1/2 (1/2)	2/2 (0/2)	6/13 (6/13)	2/3 (1/3)	1/1 (0/1)	14/23 (8/23)
9c9afd6b77d8d3a66a2db2d2cf0b94b3	3/3 (0/3)		1/2 (1/2)	2/2 (0/2)	6/13 (6/13)	2/3 (1/3)	1/1 (0/1)	15/24 (8/24)
Ssucl (MD5)								
f0bf007b3d2580297b208868425e98c7	6/9 (2/9)		1/1 (0/1)	1/1 (0/1)	11/22 (8/22)		0/2 (2/2)	19/35 (12/35)
c5a2d14bc52f109a06641c1f15e90985	7/10 (2/10)		1/1 (0/1)	1/1 (0/1)	12/19 (4/19)		1/3 (2/3)	22/34 (8/34)
Dougalek (MD5)								
95a04cfc5ed03c54d4749310ba29dda9	2/2 (0/2)		2/2 (0/2)	2/2 (0/2)	10/18 (4/18)			16/24 (4/24)
91d57eb7ee2582e0600f21b08dac9538	3/3 (0/3)							3/3 (0/3)
SUMMARY	37/53 (10/53)	5/7 (1/7)	6/8 (2/8)	8/8 (0/8)	86/136 (34/136)	600/641 (5/641)	3/7 (4/7)	745/860 (56/860)

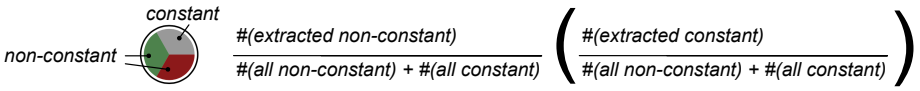

Table 9: Recall-Evaluation of HARVESTER. Green slices: amount of logging points with non-constant values where a dynamic analysis is necessary for value extraction. Red slices: amount of missing logging points. Grey slices: amount of logging points with constant values where no static/dynamic analysis is necessary. Fraction next to circle: fraction of successfully extracted logging points for non-constant values. Fraction in brackets: fraction of successfully extracted logging points for constant values.

Table 9 shows the evaluation results for logging points from the categories *URI*, *Webview*, *SMS Number*, *SMS Text*, *File*, *Reflection* and *Shell Commands*. The results for each malware sample in each category are represented as circles. Grey slices indicate the fraction of logging points that use constant values, which can be read off directly, and where consequently no backward-slicing and dynamic execution is necessary. Though the complexity of HARVESTER is not necessary to extract such constant values, HARVESTER discovers constant values at once. Green slices indicate the fraction of logging points with non-constant values for which HARVESTER was able to successfully retrieve at least one value. Red slices indicate the amount of missing logging points for which HARVESTER could not find a runtime value. The fraction directly next to the circle indicate the fraction of successfully extracted (non-constant) logging points, where the fraction in brackets show the fraction of successfully extracted logging points for constant values.

Table 9 shows two major facts: First, only 6.5% (bottom right corner $\frac{56}{860}$) of the extracted logging points contained a constant value. This confirms that a naive approach that only extracts constant values is not sufficient for our representative set of current malware. Furthermore, the table also shows that HARVESTER has a very high detection rate, since *green slices are bigger than the red slices* (bottom right corner).

In summary, the table shows that, averaged over all categories, HARVESTER detects at least one value for 86,6% (bottom right corner $\frac{745}{860}$) of all logging points. The fraction of missed logging points is due to HARVESTER's limitations (see Section 4.5) such as the lack of support for inter-component communication. HARVESTER is even able to cope with the anti-analysis techniques used by the Pincer malware family where it successfully extracts the SMS number and message, URIs, shell commands and various file accesses. The small fraction of missed logging points is mainly caused by HARVESTER's limitations, which will be discussed in Section 4.5.

We then used those apps, for which at least one value of interest was discovered, to assess HARVESTER's precision and recall. Through manual inspection we were able to confirm that *all* values discovered by HARVESTER are actual runtime values, i.e., that HARVESTER has a precision of 100% on this data set. We furthermore evaluated the recall of the extracted SMS numbers, SMS messages and shell commands of our test data since those values are among the most important ones in a malware investigation. With the help of CODEINSPECT's interactive bytecode debugger (see Section 6.1.3), an independent ethical hacker manually reverse engineered and confirmed that HARVESTER extracted all runtime values for these categories. In other words, in those experiments also HARVESTER's recall is 100%.

HARVESTER was configured with a timeout of 10 minutes. This timeout caused the execution to abort in less than 1% of all cases. Dummy values due to cut-offs during the slicing (see Section 4.3.2.1) only needed to be inserted in about 1% of all cases as well.

RQ2: How does the recall of HARVESTER *relate to existing static- and dynamic-analysis approaches?*

We next compare HARVESTER with purely static and purely dynamic approaches for automatically extracting values of interest from malicious applications.

STATIC ANALYSIS We compared HARVESTER with SAAF [Hof+13], a static approach for identifying parameter values based on a backward slicing approach starting from a method call. This method is similar to the static backward-analysis part in HARVESTER but uses traditional slicing. HARVESTER was evaluated on the same 6,100 malware samples as SAAF was evaluated (taken from MobileSandbox [Spr+13]). The logging points for both tools were the number and the corresponding message of text messages. The results for SAAF show that the tool has some issues with certain string operations such as concatenation. Instead of the concatenated string, SAAF reports the two distinct operands. This gives only partial insight into the behavior of the application. In some cases, HARVESTER found more fragments of the target telephone number as SAAF[2]. In contrast, HARVESTER extracts the final, complete SMS numbers for all of the samples, even in cases in which SAAF did not yield any data. Furthermore, SAAF does not support extracting the texts of the SMS messages being sent since they are usually not string constants, but built through

2 e.g., number `1065-5021-80133` in sample with MD5 hash `b238628ff1263c0cd3f0c03e7be53bfd`

concatenation and string transformation. Due to its static nature, opposed to HARVESTER, SAAF cannot handle reflective calls with obfuscated target strings either. We further evaluated SAAF on current Android malware taken from Table 9 including the most sophisticated Android malware families: Obad, Pincer, Ssucl and Dougalek. SAAF was configured to extract values of interest for reflective method calls, SMS numbers and SMS messages. The tool was not able to extract any value of interest for Obad, Pincer and Ssucl. For Dougalek, SAAF found the same SMS numbers as HARVESTER, but was not able to extract SMS messages. The SMS numbers can be extracted in a static way (static backward slicing) since no obfuscation is applied to the constant string values. In summary, this shows that hybrid approaches such as HARVESTER can handle current malware samples more effectively than purely static ones like SAAF.

DYNAMIC ANALYSIS Extracting values of interest can also be achieved by executing the app and applying code coverage techniques [Goo14a; MTN13; Hao+14; CNS13; Ana+12] that try to reach the statement of the logging point. To evaluate HARVESTER on dynamic approaches, we randomly chose a set of 150 samples from 18 malware families from the Malware Genome Project [ZJ12]. We compared HARVESTER's recall with 5 different state-of-the-art testing-based approaches that were publicly available to us and could be setup with reasonable effort: Google's Monkey [Goo14a], PUMA [Hao+14], AndroidHooker [Andb], DynoDroid [MTN13] and a naive approach that starts the app, waits for 10 seconds and quits the app. Unfortunately, we were not able to test Acteve [Ana+12] and SwiftHand [CNS13] on our samples due to tool-internal issues. Google's Monkey [Goo14a] was set we to run with at least 1,000 randomly-generated events per app that were limited to normal user interactions (click, swipe, navigation button use).

The goal was to find the telephone numbers to which SMS messages are sent (all 150 samples contained at least one API call for sending SMS messages). To count how many logging points the dynamic testing tools reached, we instrumented the malware samples' bytecode to create a log entry directly before sending the message. After running the testing tools, we evaluated the log output taken from the Logcat tool. All tests were carried out on an Android 4.1 emulator (API version 16).

Table 10 shows that HARVESTER's recall is around four to six times higher than the one of current state-of-the-art dynamic analysis approaches. One reason for the particularly poor recall of the existing dynamic testing tools are emulator-detection techniques. These checks prevent the tools (which run the potentially malicious apps on an emulator for security reasons [Spr+13; Lin+14]) from ever reaching a logging point in most malware samples.

Approaches	total logging-points covered
Simply open and close app	14.1%
Monkey	15.6%
PUMA	17.3%
AndroidHooker	16.2%
Dynodroid	22.3%
HARVESTER	83.4%

Table 10: Measuring recall of HARVESTER in comparison to state-of-the-art dynamic testing tools

```
1  public void onStart(Intent intent, int i)
2      ContentResolver cr = getContentResolver();
3      Cursor contacts = cr.query(CONTENT_URI, null, ...);
4      SmsManager sms = SmsManager.getDefault();
5      if (contacts.getCount() > 0) {
6        do {
7          int colIdx = contacts.getColumnIndex("data1");
8          String telNo = contacts.getString(colIdx);
9          sms.sendTextMessage(telNo, null, "I take pleasure in hurting small animals, just thought you
       should know that", ...);
10       } while (contacts.moveToNext());
11       String number = getNumber();
12       sms.sendTextMessage(number, null, "text", ...);
13     }
14 }
```

Listing 9: "DogWars" game from Malware Genome Project

As an example for such an emulator check, Listing 9 shows malicious code extracted from the "DogWars" application. It accesses the user's contact database in line 3. Only if contacts are available on the phone (line 5), the app sends out the premium SMS message (line 12). When a dynamic tool runs the app on an emulator, the contact database is usually empty and the logging point for sending SMS messages is thus never executed. As our results confirm, such behavior is common among modern malware applications. Since such checks, however, do not influence the target telephone number, HARVESTER simply removes the respective condition and correctly retrieves the number *73822*. Note that the taunting text messages (line 9) get sent to every telephone number in the user's address book and are thus data-dependent on the environment (i.e., the contact database). Thus no fixed target phone number can be retrieved by any tool. In such cases, HARVESTER reports a constant string with information about the source (e.g., contact database information). Many malicious applications such as the *GoldDream*, *BaseBridge*, and *BgServ* malware families, as well as the *DogWars* app, perform their malicious activities in a background service that is started only after the phone is rebooted. Apps from the *GPSSMSSpy* family act on incoming SMS messages. To obtain the respective runtime values, traditional dynamic approaches must generate such external events and restart the phone. HARVESTER instead directly executes the code slices containing the logging points and thus does not need to emulate these events.

To overcome simple environment checks, AndroidHooker [Andb] and Dynodroid [MTN13] first prepare the emulator with fake "personal user data" such as contacts. Only afterwards, they install the application and exercise it. Both also send external events such as incoming SMS messages and AndroidHooker even reboots the emulator during the test to trigger actions that only happen at boot time. AndroidHooker was able to reveal the premium SMS message in the *DogWars* app, but does not solve the code-coverage problem in general. For instance, it still fails if the malicious code is only executed after receiving a command from a remote server, such as in the *GoldDream* malware family. Due to such problems, AndroidHooker only covered 16.2% of all logging points. In only 10.67% of all apps it covered any logging point at all—a marginal improvement over running Monkey as is. In summary, these results show that current state-of-the-art testing tools are not sufficient for revealing malicious behavior of current state-of-the-art malicious applications. HARVESTER succeeds in all cases, as the conditional checking for the server's command is not part of the slice that HARVESTER computes, and the code containing the logging point is executed directly and unconditionally.

Furthermore, tools such as Monkey can only improve code coverage by triggering interactions in the user interface. Some malware apps from the *GPSSMSSpy* family, however, contain a broadcast receiver that directly leaks incoming SMS messages and which is completely distinct from the UI. While Monkey never executes the respective code, HARVESTER directly invokes the slice containing the data leak regardless of its original position in the code.

As a summary, dynamic tools only reach a small fraction of all logging points for these malware samples. It is worth mentioning that a naive approach that starts an app, waits for ten seconds and closes the app, produces similar results (first line in table) as Google's Monkey approach. HARVESTER, on the other hand, covers 83.4% of all logging points and thus shows a much higher recall.

RQ3: How efficient is HARVESTER*?*

App Stores such as the Google Play Store receive thousands of new or updated Android apps per day [Sta14], which they need to check for malicious behavior. Therefore, one requires fast tools, which scale to the size of the market. We tested HARVESTER on our full set of 16,799 malware samples (which includes all samples from the previous sections). We configured HARVESTER with two logging-points (SMS phone numbers and the respective text messages) for each sending SMS API call included in the app's bytecode. We focused on SMS numbers and messages since SMS trojans are among the most sophisticated malware apps today [FS14]. With HARVESTER, one can effectively find the real values for phone numbers and text messages and compare them to known blacklists or apply existing filters for identifying scamming malware.

The performance evaluation reported in this section was run on a compute server with 64 Intel Xeon E5-4650 cores running Ubuntu Linux 14.04 with Oracle's Java HotSpot 64-Bit Server VM version 1.7.0 and a maximum heap size of 20 GB to avoid intermediate garbage collection. We used the Android ARM emulator. On average, HARVESTER took about 2.5 minutes per application. This shows that HARVESTER can be used for mass analyses and delivers results very quickly. On average over all slices in all our samples, HARVESTER had to try different values for 0.21 `EXECUTOR` flags per slice. The highest average number of `EXECUTOR` flags we encountered per slice in a single app was `1.31`.

RQ4: Which interesting values does HARVESTER *reveal?*

We next report interesting values that HARVESTER extracted from malware applications. Our analysis is based on our full sample set of 16,799 malware apps. Some of these results have already been found through earlier manual investigation by security experts. However, to the best of our knowledge, HARVESTER is the first fully automated approach that is able to reveal all of these findings.

Our exemplary findings in the malware samples can be used to improve the precision of current automatic malware detection approaches such as MAST [Cha+13], DroidAnalytics [ZSL13] or Drebin [Arp+14].

HIDING SENSITIVE METHOD CALLS A growing number of sophisticated Android malware applications such as Obad [TKG13] uses reflection to call methods identified by encrypted string constants which only get decrypted at runtime. We used HARVESTER to recover the targets of these reflective method calls and found two popular obfuscation pat-

terns. In a first scenario, only sensitive API calls, such as `getSubscriberID`, `getDeviceId`, or `sendTextMessage` are obfuscated, which is likely to be the result of a manual obfuscation to hinder human analysts or automatic tools that looks for sensitive API calls such as CHABADA [Gor+14]. In the second scenario, all method calls are obfuscated, even non-critical ones such as `StringBuffer.append()` or `String.indexOf()`, which is most likely the result of automatic obfuscation tools such as DexGuard [Tec14]. In some applications even the reflective calls themselves were again called via reflection to produce a multi-stage obfuscation. HARVESTER is able to extract the called method as well as the concrete parameter values of the invocation in all these cases.

PREMIUM-RATE SMS AND SMS COMMAND AND CONTROL Silently sending SMS messages to premium-rate numbers is one of the most common Android malware schemes [FS14]. Depending on the provider and the malware, a single message can cost from about $3.5 to $6 [Chi12], which causes a high financial harm to the user. HARVESTER extracts many distinct premium-rate numbers from various known SMS trojan malware families such as *Pincer*. Many numbers can be found in multiple samples, making them good candidates for blacklisting. Many samples are obfuscated, requiring a tool such as HARVESTER.

Furthermore, most SMS trojans store the number of messages sent in *SharedPreferences*, a key-value storage provided by the Android framework. HARVESTER uncovers many keys like "SENDED_SMS_COUNTER_KEY" or "sendCount" used for this purpose. Some samples even use keys like "cost" for storing the total amount of money stolen so far. Based on these values, the malware decides when the next premium-rate SMS message is sent. We also found applications that contact a command-and-control (C&C) server via SMS. Since the same commands reappear in many samples, they also could be used for blacklisting. We also found spam messages, which are sent via SMS to all contacts on the phone. Most of them contained links to further malicious applications. The link can be used for a server shutdown to stop spreading further malware.

INTERESTING URIS HARVESTER is able to extract the concrete URL of *http* requests sent by applications. These URLs can give hints whether an application is malicious or not. HARVESTER extracts connections to advertisement servers, but also to many well-known C&C server URLs[3]. Furthermore, it also extracts many interesting phone-local URIs for accessing content providers, such as `content://sms`, `content://mms` or `tel://<number>` which are used by malware for reading SMS/MMS messages or initiating phone calls without user awareness [ZJ12]. In the case of the *tel* scheme, HARVESTER found the actual telephone numbers being called. In applications with advertisement libraries such as AirPush, HARVESTER revealed a lot of `market://details` URIs, which open the Play Store app to offer other apps for download.

EXECUTED COMMANDS We also used HARVESTER to extract runtime values for command-executing API methods such as `Runtime.exec()`. Applications containing `su` and `chmod` commands are likely to used for checking whether the smartphone is rooted or not. HARVESTER was able to detect such commands in the given malware set, even in the case of obfuscation.

3 `http://198.211.118.115:9081/Xq0jzoPa/g_L8jNgO.php`, `http://m-l1g.net/q.php`, and others

ENCRYPTION KEYS Some benign applications encrypt sensitive data such as chat conversations, or credit card information, before storing it locally on the phone. This encryption, however, is rendered useless if the same hard-coded symmetric key is used for all installations of the app. Interestingly, this was the case in the popular WhatsApp messenger app [Goo14b]. Since the encrypted database is stored on the SD card, malicious applications can easily access it. Once the key is known, it can be decrypted and leaked. HARVESTER can fully automatically extract the WhatsApp encryption key by obtaining the values passed to the constructor of the `SecretKeySpec` class.

4.4 APPLICATION SCENARIOS

The techniques provided by HARVESTER can not only be used as a direct feedback, e.g., what concrete URLs are part of the app, it can be also used to improve existing code analysis approaches. Especially in these cases where concrete values at certain code locations are essential for precise results. Please note, improving the current state-of-the-art for different code analysis approaches, e.g., dataflow analyses, also provides more detailed information about the insights of an application's behavior. This is essential during a malware investigation. In the following, we will evaluate the effectiveness of HARVESTER in combination with further code analysis approaches on three different application scenarios. As first, we will evaluate on two static analysis problems: *improving the generation of an intra-component, inter-procedural callgraph in the context of dataflow analysis* and *improving the generation of an inter-component, inter-procedural callgraph*. Afterwards, we will elaborate on a dynamic analysis problem: *improving the recall of dynamic dataflow analysis approaches*.

4.4.1 *Intra-Component, Inter-Procedural Callgraph*

As mentioned in the previous sections, reflective method calls are one of the reasons for imprecise and most of the time incomplete callgraphs in Android applications. This reduces the precision and recall of static code analysis approaches. In this section, we evaluate in more detail whether HARVESTER - used as a pre-analysis tool - can improve the precision and recall.

4.4.1.1 *Application Scenario*

For this application scenario, we used HARVESTER's code injection feature (phase C in Figure 7) to enhance the bytecode with concrete API calls for the corresponding reflective method calls. The static code analysis approach for this evaluation is the static dataflow-tracking tool FlowDroid [Arz+14b]. Especially for dataflow problems, it is necessary to have a precise callgraph for reducing the amount of false positives. Therefore, we used different Android applications that contain reflective method calls and measured how many data leaks FlowDroid detects on the original application. Afterwards, we used HARVESTER to enhance the original application and applied FlowDroid again on the new application.

⊛ = correct warning, ○ = missed leak
multiple circles in one row: multiple leaks expected

App (Obfuscated)	FlowDroid	
Enhancement	BEFORE	AFTER
Button1	○	⊛
Button3	○ ○	⊛ ⊛
FieldSensitivity3	○	⊛
ActivityLifecycle2	○	⊛
PrivateDataLeak3	○ ○	⊛ ○
StaticInitialization2	○	⊛
EmulatorDetection1	○ ○	⊛ ⊛
EmulatorDetection2	○ ○	⊛ ⊛
LoopExample1	○	⊛
Reflection1	○	⊛

Table 11: Leak detection FlowDroid on obfuscated DroidBench apps before and after value injection / slicing.

4.4.1.2 *Evaluation*

As a first evaluation on real-world applications, we chose the Fakeinstaller.AH [Rui12] malware family[4], which is known for leaking private data, but also for its massive use of reflection to hide calls to sensitive API methods. On the original obfuscated sample, FlowDroid detected only 9 distinct leaks. After using HARVESTER with the option of replacing reflective calls with their respective actual callees, FlowDroid detected 26 privacy leaks, almost three times as many as before. These 26 leaks included stealing the IMEI or phone number via SMS.

To evaluate in more detail how HARVESTER improves the precision and recall of existing tools on obfuscated applications, we tested FlowDroid on ten randomly-picked applications from DroidBench [Arz+14b] which we obfuscated using DexGuard [Tec14]. All API method calls were replaced with reflective calls on encrypted strings. Table 11 compares the detection rate of FlowDroid on the obfuscated applications without applying HARVESTER (*BEFORE* - column 2) to the respective detection rates after applying HARVESTER (*AFTER* - column 3). These results show that FlowDroid was initially not able to detect any leak in the obfuscated apps. After de-obfuscating the apps with HARVESTER through runtime-value injection, FlowDroid found the same leaks as in the un-obfuscated original version. In "PrivacyDataLeak3", FlowDroid always misses one of the two leaks, even in the original, un-obfuscated file, for reasons unrelated to the work presented in this thesis.

4.4.2 *Inter-Component Callgraph*

Apart from reflective method calls, inter-component communications are another reason for imprecise callgraphs of Android applications. An Android application usually consists of different components that communicate with each other via intent messages (see Section 2.1.1). The same applies for communications between applications. For static analysis

4 Sample MD5: 38a9ed0b5577af6392096b4dc4a61e02

approaches, it is important to precisely determine these communication links for a precise construction of the callgraph.

One of the major problems in the construction of inter-component callgraphs is the identification of concrete links between components, i.e., what components can communicate with each other. All intent-based communications in Android are realized through the Android system (see Section 2.1.2), which determines what component to start. In case of static analysis approaches, this intent resolution process needs to be realized in a static way by determining values for intents, intent-filter and URIs [Oct+16] (ICC values). These values are mainly string values. Values for intent-filters are usually provided in the AndroidManifest file, which can be parsed in a straightforward process. However, dynamically created intent-filters, intents and URIs need some form of code analysis to extract the values. If an analysis is not able to extract this information with perfect precision, it can result in over-approximations. In case of an intent and its corresponding intent filter, it is important to determine all concrete values of the intent and intent filter. For instance, if an analysis is able to determine the *action* field, but not the *category* field of the intent object, the intent resolution process may not be able to determine the correct intent filter, resulting in considering too many possible intent filters that are not the correct ones. In the worst case, a callgraph for a single application must consider all components inside the application as potential links. The problem gets even worse in inter-application communications. In such cases, different components inside the individual applications need to be additionally considered. This results in callgraph constructions that need too much memory and an analysis on top of it, e.g., dataflow analysis, would be too time-consuming.

The main problem for constructing precise inter-component callgraphs boils down to the problem of identifying concrete runtime values at certain code locations (e.g., of an intent). HARVESTER provides a solution for this and we therefore evaluate its effectiveness in the following.

4.4.2.1 *Application Scenario*

In this application scenario, we wish to examine whether HARVESTER's output can improve the state-of-the-art approaches that focus on identifying links between components in Android applications, or more precisely, whether it can reduce the amount of edges in an inter-component based callgraph.

We evaluated HARVESTER against two other state-of-the-art approaches called IC3 [Oct+15] and Primo [Oct+16] that are currently available for inter-component based callgraph constructions.

IC3 is a static constant string propagation approach that works in two stages. It first gathers the dataflow facts of constant string values and applies them to a constraint solver called COAL (constant propagation language). The task of the solver is to determine regular expressions that satisfy the constraint. For instance, if the static propagation approach was only able to determine `com.a` as package name, but the complete package name would be `com.a.b.c` of a component, the solver will return `com.a.*`. The constant string propagation analysis is inter-procedural, context-sensitive, flow-sensitive and field-sensitive. With regards to IC3's analysis and its limitations, it is sound to assume that the results (regular expressions for strings) of the analysis are either the correct information of an intent object or at least over-approximates the current values in such a way that the correct values are part of the regular expression.

After the extraction of the intent object information, it is also necessary to determine the concrete links of the components, which is realized in the same way as Android's intent resolution algorithm (see Section 2.1.2).

PRIMO One of the main limitations of IC3 is the fact that statically inferring ICC values with a static solver is in many cases too computationally expensive or even not possible in cases of values that are generated during runtime. Therefore, many ICC values are over-approximated (.*). Primo tries to improve this situation with a probabilistic approach that takes as input the results of IC3 and tries to determine, based on a probabilistic model, possible links between the components. Their main goal is to reduce the links that are produced by IC3 in combination with the intent resolution process. However, their approach is based on a probabilistic approach with some assumptions [Oct+16] and does heavily rely on the results produced by IC3.

HARVESTER Since HARVESTER's focus, similar to IC3, is only on the extraction of runtime values, we extended it with Android's intent resolution rule set. This extension takes as input concrete intent information that is extracted by HARVESTER and provides the corresponding callgraph links.

In all of these approaches, it is important to avoid considering links between components that can never occur during runtime. The matching precision, i.e., the identification of the correct intent object information, determines the precision of the overall analysis. Its influence on analysis precision is similar to the influence of the callgraph construction process in inter-procedural program analyses: an imprecise callgraph results in an overall imprecise analysis.

4.4.2.2 *Evaluation*

In the following evaluation, we focus on links between intents and intent-filter since they are one of the major problems for the over-approximation in current inter-component callgraph construction approaches [Oct+16]. A link between an intent and an intent-filter is a *true positive* if the real value of an intent object matches an intent-filter, i.e., those components that would be determined by the Android system. On the other hand, a link is a *false positive* if the value of an intent i inferred by an analysis matches an intent-filter, even though the real value of i does not. A link is considered to be a *false negative* if the real value of an intent object matches an intent-filter, but the corresponding link is not inferred by an analysis.

The evaluation was conducted on the IC3 [Oct+15], Primo [Oct+16] and HARVESTER approaches. IC3 and HARVESTER have similar goals, namely extracting values at specific code locations. IC3 realizes this with a static approach and HARVESTER with a hybrid approach. Since the outputs of both approaches are values of intent objects, we additionally implemented an intent resolution algorithm that determines the links between an intent and an intent-filter based on the inferred values. The algorithm is based on Android's intent resolution process, which is informally described online [Appf] and formally in related work [Oct+16]. Throughout the evaluation, we refer to HARVESTER but mean the combination of HARVESTER with the intent resolution algorithm. The same applies for IC3.

Unfortunately, there is no ground truth for real-world Android applications that describe the correct set of ICC links. At the beginning of this evaluation, we started with a manual

investigation, e.g., reverse engineering, to determine the ground truth, but found that it is not possible to complete this task with perfect precision for many hundred applications in a reasonable amount of time. Therefore, we decided to evaluate HARVESTER against IC3 and HARVESTER against Primo, separately. Since HARVESTER extracts runtime values in a hybrid way, it is easier to validate the results with IC3 and Primo.

In all experiments, the cut-offs of HARVESTER were 3 for caller-slicing and 10 for callee-slicing (see Section 4.3.2.1) and the timeout was set to 20 minutes, similar to IC3. Primo runs with its default settings. The approaches were evaluated on 227 malicious samples from VirusTotal[5] and 206 samples from Google Play Store (crawled date: 2014) including Android system apps. In total, we used 434 Android apks.

In the following, we consider sets of links that the individual approaches inferred. L_{IC3}, L_{Primo} and L_{HV} are sets of links between an intent and an intent-filter, correspondingly for IC3, Primo and HARVESTER.

HOW WELL DOES HARVESTER PERFORM IN COMPARISON TO IC3?

IC3 is a static approach that infers regular expressions for intent object values. Due to its static nature, it can be the case that certain fields of an intent object contain wildcards, such as "com.a.*" for the component field. This may lead to over-approximations during the intent resolution process. Therefore, as a first step, we evaluated the occurrence of ICC values that contain at least one wildcard for the action, data or category field. Our evaluation shows that 67% (all applications containing benign and malicious apps) of all intent objects contain at least one wildcard for their fields. This shows that many links between components may be false positives. In the following, we exclude the remaining 33% of intent objects for which IC3 was able to infer all concrete field values. These are all cases where IC3 was able to extract the concrete object intent and HARVESTER cannot perform better.

Table 12 shows the results of our comparison. We distinguish seven different cases (first column) that include different scenarios we want to evaluate. Note that we verified that every link in L_{IC3} and L_{HV} fits in exactly one case, i.e., no double counts and no additional cases are necessary. Furthermore, we evaluated benign, malicious and all apps separately to verify whether there are any differences. In the following we explain the meaning of each case and the results in detail. All cases are evaluated with the assumption that all intent objects extracted by HARVESTER contain either correct fields, i.e., every extracted value for a field is a real value, or wildcards, i.e., those cases where HARVESTER was not able to extract a runtime value. We verified the correctness of these cases by hand and can guarantee that this is true in all cases.

CASE 1 measures how often IC3 over-approximates ICC links. If HARVESTER's links are a proper subset of those from IC3, we can measure this fact. We explicitly excluded those cases where HARVESTER is not able to identify a link ($L_{HV} = \emptyset$), since we evaluate this in *Case 2*.

Line 1 of Table 12 shows that there are no big differences between over-approximations in the cases of benign, malicious and all applications. For the case "ICC links between all applications", including intra-application links, we can see that IC3 over-approximates in 32,45% of the cases. In other words, HARVESTER is able to reduce

5 Free service that analyzes suspicious files and URLs and facilitates the quick detection of malware. Download date: January 2016

	Case	ICC in Benign (%)	ICC in Malicious (%)	ICC in all apps (%)
1	$L_{HV} \subset L_{IC3} \wedge L_{HV} \neq \emptyset$ (over-approximation by IC3)	31.01	32.42	32.45
2	$L_{HV} = \emptyset$ (HARVESTER extracted value, but no ICC link available)	15.5	16.47	15.37
3	$L_{IC3} \subset L_{HV} \wedge L_{IC3} \neq \emptyset$ (over-approximation by HARVESTER)	1.69	6.66	4.44
4	$L_{IC3} = \emptyset$ (IC3 extracted value, but no ICC link available)	0.05	0.17	0.12
5	$L_{HV} = L_{IC3} \wedge L_{HV} \neq \emptyset$ (same links)	51.75	44.28	47.63
6	$L_{HV} \cap L_{IC3} = \emptyset \wedge (L_{HV} \neq \emptyset \wedge L_{IC3} \neq \emptyset)$ (all links are disjoint)	0.00	0.00	0.00
7	$L_{HV} \cap L_{IC3} \neq \emptyset \wedge \neg(L_{HV} \subseteq L_{IC3} \vee L_{IC3} \subseteq L_{HV})$ (parts of the links are disjoint)	0.00	0.00	0.00

Table 12: Comparing HARVESTER with IC3

the amount of links in inter-component callgraphs by 32,45%. This is a significant reduction, which shows that hybrid approaches can perform much better when extracting correct intent objects, compared to purely static ones.

The reason for this result is the fact that IC3 was either not able to extract any information about the intent, i.e., yielding wildcards for all intent fields, or it extracted an intent object that contained too many wildcards for its fields.

CASE 2 is a special case, since it considers those cases where HARVESTER is able to extract values for the intent object fields, but there is no ICC link to other components. Our evaluation showed that this is the case in 15,5% of benign and 16,47% of malicious apps (see line 2).

A closer look into the reasons revealed interesting facts. In the following, we explain the major cases we identified.

There are cases where *explicit* or *implicit* intents are implemented in the code, but no corresponding intent filter exists. There are two sub-cases, first intra-application communication links (inter-component communication within an application) and second inter-application communication links (inter-component communication between applications). Reasons for the former are cases were apps include additional code that gets not executed. These applications contained code that cannot get executed since there were no intent-filters defined (no component declaration in the AndroidManifest). One possible reason for such apps are free-version apps that mistakenly contain code from the paid version (more functionality), but the AndroidManifest is designed for the free-version app (no declaration of intent-filters). There are also cases where application developers falsely defined an intent filter, such as `UnityPlayerNativeActivity` instead of `UnityPlayerActivity`. Reasons for inter-application communication are apps that need to communicate with other apps, e.g., the YouTube app, but these apps where not part of our corpus. Therefore, we were not able to identify the corresponding link.

Intent-filters for broadcast-receiver can be, apart form the static declaration in the AndroidManifest, also dynamically registered in the application's code (see Section 2.1.2). Unfortunately, we did not consider these cases in our evaluation and were therefore not able to identify the corresponding ICC link. This is subject to future work.

Especially for malicious applications, there are apps that want to exploit security vulnerabilities in system applications. These are vulnerabilities that are based on the fact that a system application usually has higher privileges as a user app. In particular, we found cases where system apps accepted intents in SMS receiver services. This can be either used to fool the user that she received an SMS, even if she did not or if the corresponding service contains a further vulnerability, e.g., SQL injection, further damage can be produced. However, since our corpus of apps consisted of the latest versions of Android system apps, these kinds of vulnerabilities were already fixed and no ICC link was determined. This is one of the reasons why the percentage of malicious applications (column 3 line 2) is slightly higher than the one of benign applications (column 2 line 2).

Another reason why malicious applications contain missing ICC links is the fact that HARVESTER has some limitations with apps that are protected by an integrity check (will be explained in Section 4.5). For instance, HARVESTER was able to extract a

value (`"#V#TW- VC%RK(X8\EW&ux*&‘"`) for an action field, but the value is encrypted due to an integrity check. However, these cases are covered by FUZZDROID, which will be explained in Chapter 5.

CASE 3 covers those cases where HARVESTER produces more ICC links than IC3. The relatively low percentage of 4,44% for all apps is caused by HARVESTER's limitations, which will be described in more detail in Section 4.5. More concrete, there are cases where IC3 is able to extract parts of a field, resulting in values with a wildcard, e.g., `"http://www.bitinstant.com?addr=(.*)"` for the data field. In this particular case, another string (`value2`) gets appended to the string: `"http://www.bitinstant.com?addr="` `+ value2`. However, due to HARVESTER's limitations, HARVESTER is sometimes not able to extract the complete value (e.g., `value2`), even if `"http://www.bitinstant.com?addr="` is part of the slice. HARVESTER cannot return a concrete value (e.g., missing information about `value2`), which results in a wildcard for the value. Therefore, IC3 is able to extract at least parts of the value, while HARVESTER is not, resulting in the case that ICC produces less links.

CASE 4 is similar to *Case 2*, with the difference that we evaluated IC3 on those cases that do not find an IC3 link. The very low percentage of 0.12% in comparison to HARVESTER's 15,37% (line 2 column 4) reveals that HARVESTER is able to extract more concrete field information than IC3. This can be inferred since *Case 4* and *Case 2* covers only those cases where IC3 or HARVESTER extracted concrete values for the intent object fields instead of a wildcard. If only a wildcard would have been extracted, all possible components would have been connected to the intent object (no empty set).

A closer look into the intent objects that were responsible for no ICC links revealed that IC3 covered similar issues explained in *Case 2*. Furthermore, we also found a possible bug in IC3 since we verified that `"com.pokercity.ddz.NULL-CONSTANT"` is not a valid account name.

CASE 5 measures how often both, IC3 and HARVESTER determine the exact same ICC links. Table 12 shows that this case is true in almost 50% of benign, malicious and all apps. The reasons therefore are cases where IC3 and HARVESTER extracted similar values for the intent object fields that link to the same components or where both approaches over-approximated for all intent object fields. We investigated those cases where HARVESTER was not able to extract any intent object information and conclude that these cases exist due to HARVESTER's limitations (see Section 4.5). Improving the limitations in subject for future work.

CASE 6 AND CASE 7 show that there are no odd cases that need to be further investigated.

As a next step we further evaluated the frequency of ICC link reduction/over-approximation produced by HARVESTER's over IC3.

Figure 10 shows a plot that contains all links from *Case 1* and *Case 3* applied on all applications. We only took *Case 1* and *Case 3*, since these cases contain ICC links where HARVESTER either reduces the amount of links or adds more in comparison to IC3. One can see that for the majority of intent objects, approximately 1200, HARVESTER is able to reduce more than 1000 ICC links per intent object. On the other hand, for a smaller amount of intent objects, approximately 200, HARVESTER produces 1000 more ICC links than IC3.

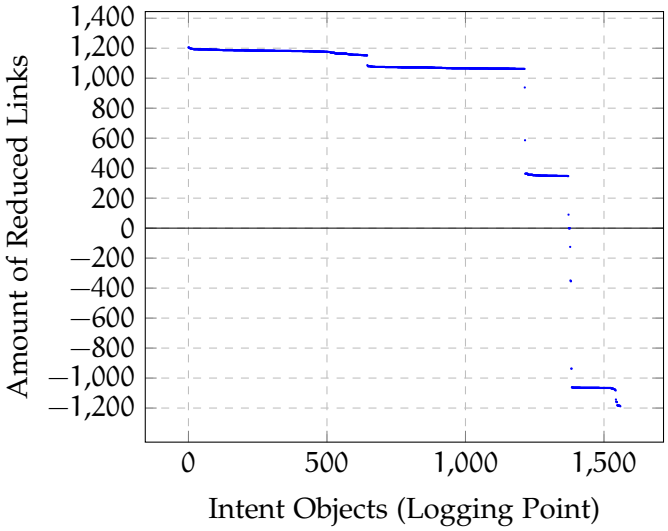

Figure 10: HARVESTER's reduction of ICC links produced by IC3

In total, HARVESTER is able to reduce 1,425,438 links (32,45%) and adds 192,520 (4,44%) more links to the inter-component callgraph.

As a summary, we conclude that a pure static determination of ICC links between benign, malicious and even in both cases produces too many false positives. In total 32,45% for all applications. Instead, extracting runtime information for intent object fields in a hybrid way, as proposed by HARVESTER, reduces false positives, which results into much smaller callgraphs. However, we state that a static analysis should be used as a pre-analysis for determine those cases where all intent object field information is provided in such a way that it is statically easily to extract. In our evaluation, this was the case in 33% of all intent objects. This step is necessary, since a static analysis is much faster than a hybrid approach.

HOW WELL DOES HARVESTER PERFORM IN COMPARISON TO PRIMO?

Primo is based on a probabilistic model and there is no guarantee that Primo's results are free of false negatives. This was our first part of the evaluation. We were interested in those cases where Primo produces false negatives ($\exists x \in L_{HV} : x \notin L_{Primo}$). Interestingly, we found 13% of all ICC links to be false negatives. We manually verified all of them and found interesting cases for which the probabilistic approach has some weaknesses. These are new findings, which were not identified in Primo's [Oct+16] work.

EXPLICIT INTENTS There are cases where IC3 is not able to identify the component value of an intent object, but HARVESTER extracted a component name. Therefore, the intent object can only be explicit not implicit (see Section 2.1.2). This observation is not known from a view of Primo. It instead uses a probabilistic approach that tries to reduce the over-approximation. However, it should not be the case that the explicit intent is missing in the results (false negative). These cases were identified by us and shows that Primo reduces the over-approximation in such a way that it removes the valid ICC link from IC3's results.

IMPLICIT INTENTS We identified a similar situation for implicit intents. There are some cases where HARVESTER is able to identify the correct links between an intent and an intent-filter, but Primo's outcome misses such links, resulting in false negatives.

After excluding the false negatives from our initial sets, we continued evaluating the remaining in the same way as IC3 (please see previous paragraph). The results are provided in Table 13.

	Case	ICC in Benign (%)	ICC in Malicious (%)	ICC in all apps (%)
1	$L_{HV} \subset L_{Primo} \wedge L_{HV} \neq \emptyset$ (over-approximation by Primo)	33.1	36.9	35.85
2	$L_{HV} = \emptyset$ (HARVESTER extracted value, but no ICC link available)	16.0	17.96	16.35
3	$L_{Primo} \subset L_{HV} \wedge L_{Primo} \neq \emptyset$ (over-approximation by HARVESTER)	0.00	0.00	0.00
4	$L_{Primo} = \emptyset$ (Primo has no ICC link)	0.00	0.05	0.03
5	$L_{HV} = L_{Primo} \wedge L_{HV} \neq \emptyset$ (same links)	50.9	45.15	47.81
6	$L_{HV} \cap L_{Primo} = \emptyset \wedge (L_{HV} \neq \emptyset \wedge L_{Primo} \neq \emptyset)$ (all links are disjoint)	0.00	0.00	0.00
7	$L_{HV} \cap L_{Primo} \neq \emptyset \wedge \neg(L_{HV} \subseteq L_{Primo} \vee L_{Primo} \subseteq L_{HV})$ (parts of the links are disjoint)	0.00	0.00	0.00

Table 13: Comparing HARVESTER with Primo

Please note that one cannot directly compare IC3's Table 12 with the one of Primo (Table 13), since we further removed ICC links in Primo's case (false negatives).

CASE 1 includes those cases where Primo over-approximates ICC links. It is interesting to see that Primo's probabilistic approach is not very effective for 33,1% of benign applications and 36,9% of malicious applications. One of the reasons hereby is the fact that IC3 (see previous paragraph) extracted a lot of wildcards for intent object fields, which directly affects Primo's probabilistic approach. This shows the importance of precise runtime value extraction approaches. HARVESTER is one of them.

CASE 2 is equal to *Case 2* of Table 12, with the only difference that Table 13 includes less links (removed false negative cases).

CASE 3 shows that HARVESTER did not over-approximate ICC links in the comparison with Primo's results.

CASE 4 is a special case where Primo was not able to detect a link. It is important to note that in all of these cases, IC3 provided intent object values, which included enough information to correctly link to the corresponding component. However, Primo's probabilistic approach removed those links resulting in false negatives. Please note that these cases were originally not counted in our removal of false negatives since $\exists x \in L_{HV} : x \notin L_{Primo}$ did not hold.

CASE 5 covers those cases where Primo has exactly the same edges as HARVESTER. Reasons therefore are already mentioned in *Case 5* of IC3's comparison. In addition to that, similar to *Case 1* of Primo's comparison, IC3 extracts a lot of wildcards for the different intent object fields, which again negatively influences Primo's results where Primo considers all links.

CASE 6 AND CASE 7 show that there are no odd cases that need to be further investigated.

As a next step, we further evaluated the frequency of ICC link reduction produced by HARVESTER over Primo.

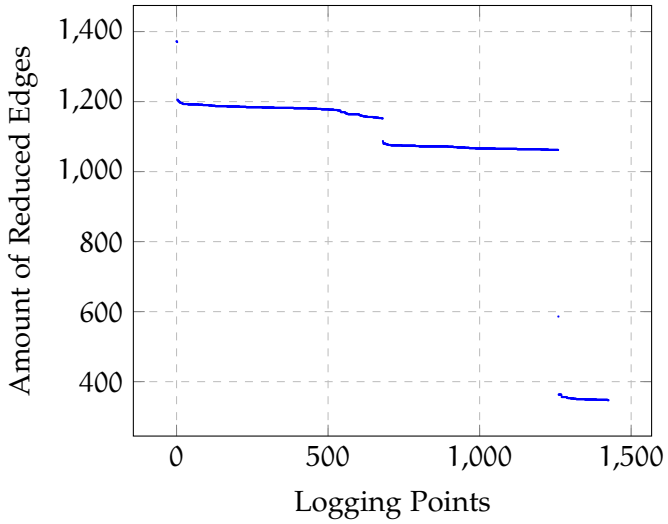

Figure 11: HARVESTER's reduction of ICC links produced by Primo

Figure 11 shows that for the majority of intent objects, approximately 1200, HARVESTER reduces more than 1000 edges for a single intent object. In total, HARVESTER reduced 1,481,075 links in the inter-component callgraph. The results are very similar to those in Figure 10 due to the fact that IC3 was not able to extract non-wildcard values for the intent object fields.

As a summary, Primo's probabilistic approach for reducing false positives heavily depends on IC3's output, which showed that Primo was not able to reduce many links due the imprecise results of IC3. A more precise approach for extracting runtime values is required to improve Primo's probabilistic approach. HARVESTER is one of those approaches. Furthermore, the evaluation also revealed that a probabilistic approach as described by Primo often results in false negatives, which is a big issue for static code analysis that rely on complete inter-component callgraphs.

4.4.3 *Dynamic Dataflow Tracking*

When compared to dynamic dataflow analysis approaches, static dataflow analysis approaches have the advantage that they have a more complete view on the application's code. However, as mentioned in the previous sections (especially in Section 4.4.1 and Section 4.4.2), different limitations prohibit the complete view. Apart from the mentioned problems of the usage of reflective method calls or the generation of inter-component callgraphs, there are further problems that inhibit a static analysis from creating a complete callgraph, for instance, applications with encrypted code parts that include sensitive API calls [AA14].

On the other side, dynamic taint tracking can only evaluate the code that gets executed. More concretely, the dynamic analysis first has to reach the code position of the source before it can start tracking the taint until it reaches a sink. As we have shown in our evaluation in Section 4.3.3, current testing approaches for Android, however, often fail to trigger the malicious behavior in current malware samples. This results in undetected data leaks.

4.4.3.1 *Application Scenario*

HARVESTER's static slicer extracts exactly the code required for computing a specific value of interest. Afterwards, only this code runs on an emulator or a real phone. Most importantly, the reduced code executed by HARVESTER does not include any emulator checks or other techniques targeted at hindering dynamic analysis. Furthermore, no user interaction with the application is required anymore, eliminating code coverage issues with existing input-generation approaches. Running existing off-the-shelf dynamic analysis tools not on the original APK, but on the reduced APK (see phase B in Figure 7) created by HARVESTER can thus greatly improve their recall as we show in this section. In this case study, we compare the recall of the well-known dynamic taint tracker TaintDroid [Enc+10] on the original APK file and on HARVESTER's reduced version.

4.4.3.2 *Evaluation*

In an approach similar to Anubis [Lab14], we ran TaintDroid 4.1 inside the emulator on the Tapsnake [ZJ12] malware sample[6], which steals location data only after a delay of 15 minutes [Yan+13]. On the original malware, the analyst needs to know that she has to wait this time. With the app reduced by HARVESTER's slicing approach, TaintDroid reports the leak instantly, without any UI interaction.

As a second evaluation, we took 10 randomly chosen examples from DroidBench (similar to Section 4.4.1) and obfuscated them with DexGuard [Tec14]. Table 14 compares the recall of TaintDroid on the obfuscated apps (BEFORE in column 2) with the recall after using HARVESTER's value injection (AFTER in column 3). In the original app, TaintDroid missed leaks depending on user actions such as in "Button3". On apps containing emulator-detection checks it failed as well. When running the slices extracted by HARVESTER (see "Reduced APK" in Figure 7), both types of leaks are found fully automatically without any user or machine interaction. The remaining missing leaks occur due to TaintDroid not considering Android's logging functions (e.g., `Log.i()`) as sinks, as we confirmed with the authors of TaintDroid.

6 Sample MD5: 7937c1ab615de0e71632fe9d59a259cf

⊛ = correct warning, ○ = missed leak
multiple circles in one row: multiple leaks expected

App (Obfuscated)	TaintDroid	
Enhancement	BEFORE	AFTER
Button1	○	⊛
Button3	○ ○	⊛ ○
FieldSensitivity3	⊛	⊛
ActivityLifecycle2	⊛	⊛
PrivateDataLeak3	⊛ ○	⊛ ○
StaticInitialization2	⊛	⊛
EmulatorDetection1	○ ○	⊛ ○
EmulatorDetection2	○ ○	⊛ ○
LoopExample1	⊛	⊛
Reflection1	⊛	⊛

Table 14: Leak detection by TaintDroid on obfuscated DroidBench apps before and after value injection / slicing. Note that we did not have to interact with the app for the TaintDroid test.

```
1 String number = null;
2 if(simCountryIso().equals("DE"))
3   number = 9371;
4 if(simCountryIso().equals("XX"))
5   number = 0000;
6 sendTextMessage(number, "msg");
```

Listing 10: Path over-approximation

4.5 LIMITATIONS AND SECURITY ANALYSIS

While HARVESTER improves over the state-of-the-art significantly, like any approach it comes with some limitations. We next discuss those limitations and how malware authors could potentially exploit them. Attempting to overcome those limitations will be an interesting piece of future research.

Attacking Timeout Mechanism

To compute the values of interest, HARVESTER executes the extracted slices. Execution ends if either all values of interest have been computed, or a timeout occurs. An attacker can theoretically exploit this timeout by deliberately creating large apps with many dataflow dependencies on the values of interest. Such an attack would lead to larger slices, and hence, longer execution times per slice, making timeouts - and thus missed values - more likely. An analyst, however, can easily increase the timeout if she detects that they happen too frequently and results are poor. Additionally, one has to keep in mind that such *Data- and Control-flow obfuscations* also increase the code size and execution time of the original app. This would severely limit the practical applicability of such obfuscators.

Overwhelming the Analyst with Spurious Values

Since HARVESTER over-approximates the paths to be executed, it may yield false positives, i.e., values that cannot be computed by the original program in any given environment. The code in Listing 10 computes a different telephone number for every mobile carrier country. The code assigning the value `0000`, however, can never be reached in the original program because there is no environment with an `XX` country code. Since HARVESTER cannot make any such assumptions about the possible set of environments, it explores this path as well, reporting the spurious value `0000`. For future work, we plan to additionally add semantic checks that try to verify the validity of an environment-check (e.g., whether `if(simCountryIso().equals("XX"))` is a valid check or not) to eliminate fake environment checks.

Hiding Logging Points

HARVESTER is currently implemented for the Dalvik part of Android applications. Section 4.3.2.1 described that HARVESTER is able to handle applications containing native method calls as long as the logging point is still contained in the Dalvik code. If, for instance, an SMS message is sent by native code, this hidden call to `sendTextMessage()` cannot be used as a logging point. If an attacker hides the complete computation of the value of interest in native code and never yields the computed result back to the Dalvik layer, HARVESTER will not be able to extract these values. However, according to previous research, current state-of-the-art banking trojans [Ras+15b] use native code mainly to hide sensitive information but leak the data in the Dalvik part. In such cases, HARVESTER can extract this sensitive information, returned by the native methods, without problems.

HARVESTER can succeed, however, if the app loads Dalvik code dynamically. In such a situation, the analyst would first run HARVESTER once to obtain the dynamically loaded code (which is just another runtime value), and then once again to extract the values of interest. In the first run, the dynamically loaded code will be merged into the dex-file and in the second step the hidden logging point in the merged dex-file will be recognized and analyzed by HARVESTER. However, if an application dynamically loads code that is encrypted with the hash of the certificate of the original application, HARVESTER will not be able to extract the packed dex-file. In Android, every application gets signed with a developer certificate. HARVESTER needs to do this as well with our own certificate. Therefore, HARVESTER's hash of the certificate is different to the one of the original application and we are not able to extract the correct value. This technique is used to protect the applications integrity (see Section 2.3). In Chapter 5 we introduce a new approach that is able to address this problem.

Attacking Static Backward Slicing

Attackers could also focus on the static backward slicing. To compute a static program slice, a complete callgraph is indispensable, as with an incomplete callgraph the slices may be incomplete as well. If an app contains multiple layers of reflective calls, the slices computed by HARVESTER will be incomplete. However, since HARVESTER is able to replace reflective method calls with their original call targets (see Section 4.3.2.3), an analyst can run HARVESTER multiple times, removing one layer of reflective calls per run. In the

end, HARVESTER is able to construct a complete callgraph and, hence, a complete slice. The same technique of multiple executions can also be used if reflective calls occur in the code that computes the target of further reflective calls.

At the moment, HARVESTER does not support slices that span multiple Android components. If a value, for instance, is computed in one activity and then sent to a second one, which then contains the logging point, this value will be missed. In the future, we plan to extend HARVESTER with support for inter-component communication, by integrating HARVESTER's ICC link extraction feature described in Section 4.4.2.

Attacking Data Dependency

We assume the values of interest not to be data dependent on environment values. For current malware this proves to be a reasonable assumption. If malware developers will introduce such dependencies in the future, one could react by extending HARVESTER to detect and report such cases to a human analyst. This can be achieved with the help of a static dataflow tracking approach that tries to identify whether the logging point is data dependent on an environment value. While this approach can be attacked due to its static nature, such a detection would significantly raise the bar for an attacker. Note that HARVESTER can be applied iteratively to remove layers of obfuscation (e.g., replace reflective calls with direct method invocations). In every run, the app gets simpler and, thus, more accessible to such static detections.

Attacking the Completeness of Values of Interest

If values of interest are computed using data from external resources such as servers on the web, we assume this data to be static. If, for instance, a remote server returns different target phone numbers for an SMS scam every day, HARVESTER will only be able to recover the value of interest for the present day.

4.6 RELATED WORK

Researchers have proposed various approaches for analyzing the behavior of Android applications. Tools which simply convert the Android dex code back to Java source code such as ded [Enc+11a] or Dare [OJM12] suffer from the problem that obfuscated applications do not contain sensitive values such as URLs or telephone numbers in plain, but the analyst rather needs to reconstruct them by manually applying the deobfuscation steps that would normally execute at runtime.

The remainder of this section describes more advanced approaches that provide a higher level of automation using static, dynamic, or hybrid analysis techniques.

STATIC ANALYSIS FlowDroid [Arz+14b] or DroidSafe [Gor+15] are static taint analysis tools which determine whether sensitive information is leaked in an Android application. Due to their static nature, they cannot handle reflective calls whose target class or method name is decrypted or concatenated dynamically at runtime. CHEX [Lu+12], IC3 [Oct+15] or Amandroid [Wei+14] are static approaches that face the problem of inter-component dataflow tracking in Android applications. Just like FlowDroid, the approaches rely on

a complete call graph and thus fail if call targets are obfuscated using reflection. They would thus also benefit from our runtime value injection for a more complete analysis. SAAF [Hof+13] is a purely static tool for finding constant strings in Android applications based on backwards slicing. It does not aim at providing any runtime semantics, e.g., if an application decrypts a constant string at runtime, SAAF will only produce the original cipher text, leaving substantial work with the human analyst. DroidRA [Li+16b; Li+16a], R-Droid [Bac+16b] and an approach by Barros et al. [Bar+15] are pure static based approach for resolving reflective method calls in Android applications. Since they can only cope with applications that contain constant strings, they are not able to resolve reflective method calls for highly obfuscated Android malware such as Obad.

DYNAMIC ANALYSIS Dynamic approaches that profile runtime behavior such as Google Bouncer [OM12a] can only detect runtime values that violate the Play Store's policy (e.g., blacklisted URLs or telephone numbers) if they are actually used in API calls during the test run. Malware, however, often employs sophisticated mechanisms to detect whether it is run in an emulator or simply waits for longer than the test run lasts before starting the malicious behavior. TaintDroid [Enc+10] is a dynamic dataflow tracker, which detects leaks of sensitive information at runtime. Other techniques such as Aurasium [XSA12] inject a native code layer between the operating system and the Android application, which intercepts sensitive API calls and checks the data passed to them. Dynamic determinacy analysis [Sch+13] is an approach for identifying values that always have the same value in all executions of a program, regardless of the input values. All these approaches share the problem of only finding values in code that is actually executed, thus requiring a test driver with full code coverage. HARVESTER circumvents this problem by directly executing the code of interest regardless of its position in the original application.

HYBRID ANALYSIS TamiFlex [Bod+11] monitors reflective method calls in Java applications at runtime and injects the found call targets into the application as call edges to aid static analysis tools. It does not support Android, however, and employs no slicing. Instead, it always executes a full, single run, leaving open how full coverage of callees is to be achieved. AppDoctor [Hu+14] slices Android applications to find user interactions that lead to application crashes. AppDoctor's hybrid slice-and-run principle is similar to HARVESTER. However, AppDoctor executes the complete derived UI actions, while HARVESTER's slices only contain code contributing to the value of a concrete value of interest. AppSealer [ZY14] performs static taint tracking on an Android application and then instruments the app along the respective propagation paths to monitor for actual leaks at runtime, effectively ruling out false positives introduced by the static analysis. It then fixes component-hijacking vulnerabilities at runtime if sensitive data reaches a sink. This approach can, however, not find leaks missed by the static analysis and thus inherit the problem of reflective method calls. SMV-Hunter [Sou+14] scans for custom implementations of the SSL certificate validation in Android applications. It first statically checks whether custom validation routines are present. If so, the dynamic part attempts to trigger this code and confirm a man-in-the-middle vulnerability. The tool only supports simple UI interactions that neither span multiple pages nor require complex inputs. Rozzle [Kol+12a], a tool for de-cloaking Internet malware has a similar goal as HARVESTER, but has its limitation in triggering the malicious behavior. For instance, it cannot handle time bombs or logic bombs. Zhou et al. [Zho+15] present an approach that is, just like HARVESTER, based on

slicing and execution. They, however, execute the app inside a custom interpreter, which is also responsible for steering the execution into specific branches. As this approach completely replaces the Android OS, it requires a very precise model of the OS and its libraries. Roundy et al. [RM10] combine static and dynamic code analysis in order to make the CFG more precise in cases where malware is packed, obfuscated or dynamically loads additional code. Zhao et al. [ZAH11] provide an approach for extracting runtime values for native binaries. They also combine static backward slicing with dynamic code execution, but their extracted slice contains an unmodified code, including conditions. This results in a lack of extracting values of interest since only one path will be executed during runtime.

UI-AUTOMATION SwiftHand [CNS13] uses machine-learning to infer a model of the application which is then used to generate concrete input sequences that visit previously unexplored states of the app. On complex user interfaces, however, SwiftHand's code coverage can fall under 40% according to the numbers stated in the paper. Code that is only executed in specific environments (e.g., depending on data loaded from the Internet) might not be reached at all. Dynodroid [MTN13] instruments the Android framework for capturing events from unmodified applications, generated both by automatic techniques such as MonkeyRunner [Goo14a] and by human analysts. On average, it achieves a code coverage of 55%. Brahmastra [Bho+14] is another UI-testing tool that combines static analysis with bytecode rewriting in order to directly execute certain code statements. Since the tool relies on a complete static callgraph, it has its limitation in applications that are obfuscated with reflective method calls such as the one used in the Obad malware family. AppsPlayground [RCE13] uses an enhanced version of TaintDroid [Enc+10] for dynamic dataflow tracking. The authors changed the Android framework to additionally monitor specific API and kernel level methods. For exercising the application at runtime, they use random testing guided by heuristics leading to a code coverage of about 33%. As HARVESTER directly executes the code fragments of interest, it does not need a method for UI automation, avoiding the problem of poor coverage and recall.

4.7 SUMMARY AND CONCLUSION

In this chapter, we showed that the extraction of runtime values from applications that include anti-static and anti-dynamic code obfuscation techniques is only possible if one combines techniques from static analysis with techniques from dynamic analysis. More concretely, in order to reach a certain code location in highly obfuscated applications one needs to combine a particular variation of static program slicing with code generation and concrete dynamic code execution. This is another major contribution of this dissertation and answers the research question in the beginning of this chapter.

In addition, we have also shown that the techniques proposed by HARVESTER, together with bytecode manipulation, can be used for resolving reflective method calls in Android applications, another major contribution of this thesis. This is especially important in cases where reflective method calls are actively used as an obfuscation technique against static code analysis approaches. Furthermore, this positively influences the creation of precise static intra-component, inter-procedural callgraphs. However, with the help of HARVESTER it is also possible to minimize the amount of inter-component callgraph edges. Existing off-the-self static analysis tools that extract concrete insights about the behavior of an application, e.g., dataflow tracking approaches, immediately benefit from this

situation. They have a more complete (or more precise) view about the application's code resulting in determining more concrete insights about the behavior of an application. This was not able prior to this work since existing inter-component callgraph construction approaches created callgraphs containing too many edges (impractical). The reduction of inter-component callgraph edges is another major contribution of this work.

We further demonstrated that the technique of combining program slicing with code generation and concrete dynamic code execution also positively influences dynamic taint tracking approaches. We reduced the main problem of reaching a data leakage source position along the dataflow path until the sink's position to a minimal piece of code that immediately executes the data leakage. Existing off-the-self dynamic taint tracking approaches are able to immediately extract concrete data leakages.

5 ENVIRONMENT INFERENCE

In the previous chapters, we proposed SuSi and HARVESTER, two novel approaches that improve current malware investigations with concrete insights about the behavior of an application. However, there are different situations in which both approaches do not provide sufficient insights that would be necessary for identifying a malicious behavior in an application.

For example, one of the limitations (see Section 4.5) of the HARVESTER approach are values of interest that are data-dependent on input data. Let's consider the second argument of the `sendTextMessage()` method call (line 51 in Figure 1) in the main motivating example as a logging point, which is data-dependent on an incoming SMS message. HARVESTER will not be able to extract a value of interest, since it is not able to trigger an incoming SMS method with the format of `#s:<some text>`. A correct environment (incoming SMS message with a specific format) needs to be identified in order to generate an example for a value of interest.

Another example are C&C servers that are hosted on legitimate websites. For remaining undiscovered by a security expert, many C&C servers are hosted on legitimate websites that are either controlled by the attacker or that are not recognized by legitimate web services such as Facebook's Parse [RCH15]. In such cases, the extraction of concrete runtime values like URLs are not sufficient enough to judge whether an application contains malicious behavior or not. It would be more important to extract concrete circumstances under which the app starts communicating with the remote server, for instance. This might provide more information for judging about malicious behaviors. For instance, in our motivating example (see Figure 1), the application requires a specific incoming SMS message, needs to wait for a certain amount of time and requires at least one contact on the smartphone before an SMS message gets sent. These can be good hints for malicious activities.

Moreover, there are different security-relevant API calls that do not contain any argument for which one could extract a concrete runtime value. An example is the `AlertDialog.show()` API method. Once implemented, it shows a dialog to the user that provides arbitrary alert information. This is a very common way in benign Android applications to ask the user about taking a decision between yes or no in response of any particular action taken by the user. However, it is also a common way of malicious applications that apply a phishing attack [Ras+15b]. The attacker provides a dialog to the user that prompts her to insert her credit card credentials, for instance. In order to look legitimate, these kind of malicious applications usually scan the victim's phone for certain applications, e.g., a certain banking application [Ras+15b], and prompt a dialog that looks identical to the one of the installed application. All these conditions, under which the dialog gets prompted, provide a lot of information that helps a security analyst in identifying a malicious behavior.

Therefore, in this chapter, we address the following fundamental research question:

How can one automatically extract environment-dependencies under which a certain code location gets reached, even if an application contains anti-static and anti-dynamic code obfuscation techniques?

In this chapter, we present FuzzDroid, a framework for automatically generating an Android execution environment where an app exposes its malicious behavior.

In summary, this chapter contributes the following:

- A framework to generate Android environments that expose otherwise hidden malicious behavior,
- a set of static and dynamic analyses that provide values for circumventing various checks in malware apps,
- a search-based fuzzing algorithm that selects environment values to steer an app toward a target location and
- empirical evidence that the approach is efficient and effective for current malware apps, and that it clearly outperforms the closest existing approach.

Chapter Outline. Section 5.1 describes a motivating example that will be used for this chapter. In Section 5.2 and Section 5.3 we explain our framework in detail. Then we provide information about the implementation (Section 5.4) and evaluate our approach (Section 5.5). In Section 5.6 we describe limitations of our approach and compare FuzzDroid with related work in Section 5.7. The chapter concludes with a summary in Section 5.8.

5.1 MOTIVATION AND CONTRIBUTION

This section illustrates several key challenges for reaching a particular code location in state-of-the-art malware apps and outlines how our approach addresses these challenges. Listing 11 shows the motivating example for this chapter. It is a modified version of the main motivating example (Figure 1) containing additional environment-checks from the *Anserver* malware family.

The code shows the same functionality as the main motivating example (see Section 1.1.1); it sends an SMS message to all contacts stored on the smartphone with a text that is received from an incoming SMS message. However, this behavior gets only triggered under certain circumstances. Automatically sending SMS messages to friends in a malicious context is usually used for *spam distribution*. The malware tries to distribute further malicious applications by requesting friends to download a certain application.

To understand an app's behavior, a human analyst or an automated analysis is interested in the conditions under which the app sends an SMS message. The human can directly draw conclusions if a certain behavior only occurs under certain suspicious environments. An analysis tool can configure its sandbox for dynamic investigation with the right environment.

In the example, suppose that we mark the call to the `sendTextMessage()` method (line 45) as the target location and want to trigger an execution where the app calls this method. Reaching this target location is difficult because the app requires a particular environment to expose its malicious behavior:

```
class SMSReceiver extends BroadcastReceiver {
String hashValue = "389d90"

void onReceive(SMS sms, Intent intent) {
  String body = sms.getSMSBody();
  String certificateHash = this.getCertificate().getHash();

  //expects "#s:<spam message>"
  if(body.startsWith("#s:") {
    // integrity check whether the APK got modified
    if (certificateHash.equals(hashValue)) {
      // get MCC and MNC codes
      String mobileOp = getNetworkOperator();
      File encryptedFile = readFileFromStorage();
      File decryptedFile = decryptFile(encryptedFile);
      boolean containsMobileOp = false;
      Reader bf = new Reader(decryptedFile);
      String line;
      // checks whether file contains specific
      // mobile operator
      while((line = br.readLine()) != null) {
        if(line.equals(mobileOp)) {
          containsMobileOp = true;
          break;
        }
      }
      // targeted attack against specific network
      if (containsMobileOp) {
        String spamMessage = body.substring(2);
        Set<Contact> contacts = getAllContacts();

        //30 minutes
        wait(30 * 60000);
        // dynamic class loading, expects a dex file,
        // even though file suffix is .db
        DexClassLoader dcl = new DexClassLoader("anserverb.db");
        Class clazz = dcl.loadClass("BaseABroadcastReceiver");
        Method method = clazz.getMethod("onStart", Intent.class);
        boolean returnValue = (boolean)method.invoke(intent);
        if (returnValue == false) {
          for(Contact contact : contacts) {
            String contactNumber = contact.getNumber();
            //sends a spam message like "check out the following app:
            //www.malicious.com/malware.apk" to all contacts
            sendTextMessage(contactNumber, spamMessage);
          }
        }
      }
    }
  }
}
}
```

Listing 11: Motivating example FuzzDroid: shows that SMS messages are only sent under certain circumstances

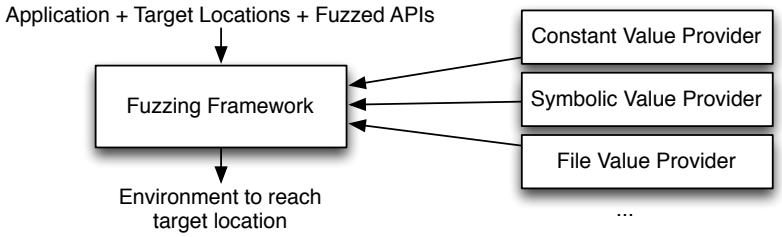

Figure 12: Overview of the FuzzDroid approach

- The app checks whether the user's network operator is part of a pre-defined list of targets (lines 13 to 28). This kind of technique is usually used in cases of targeted attacks where only specific users are attacked, e.g., only users located in a certain country.
- The behavior must be triggered by an incoming SMS message that starts with a particular string (line 9). This is a common behavior of command-and-control communications.
- The smartphone has to have an address book containing at least one contact (line 41).

The problem is compounded by the fact that the app tries to evade analysis:

- The app checks whether the APK file of the app has been modified. It obtains the signature with which the running app has been signed, hashes it, and compares it against a known hash value (line 11). Such behavior is designed to prevent instrumentation-based dynamic analyses from modifying the app.
- The app loads an additional class from a file, reflectively calls a method of the loaded class, and checks whether the method returns a particular value (lines 36 to 40). Such behavior challenges static analyses, which cannot easily reason about reflective methods calls, and in particular, about methods of classes that are dynamically loaded from an external file.

All these conditions are typically hidden in highly obfuscated code. A naive dynamic analysis that simply executes the app, possibly by sending a random SMS, does not reach the target location. Randomly fuzzing the environment is very unlikely to be successful, because multiple non-trivial conditions must be met. A purely static analysis [Fra+16] cannot easily reason about reflectively called methods, and in particular, about methods of classes that are dynamically loaded from an external file. Even a recently-proposed approach that combines static and dynamic analysis [WL16] cannot reach the target, among other reasons because it cannot deal with obfuscation through reflection.

FuzzDroid addresses the problem of extracting the constraints and finding a matching environment through directed fuzzing. Figure 12 gives a high-level overview of the approach. Given an app, a set of target locations, and a set of APIs to fuzz, the approach repeatedly executes the app while fuzzing the values returned by the specified APIs, until it finds an environment where the app reaches a target location. To this end, the approach intercepts calls of the app to APIs and modifies their return values to steer the app toward the target.

An observation crucial for our work is that a single fuzzing approach is insufficient to reach target locations in real-world malware apps. Instead, FuzzDroid consists of a generic framework and an extensible set of value providers. Each *value provider* is a static or dynamic program analysis that provides values to the fuzzed APIs. To decide which of the suggested values to use for a particular API call, we present an evolutionary algorithm-based search strategy. The strategy iteratively refines the selection of values based on feedback from previous executions.

To illustrate FuzzDroid with the motivating example, suppose we trigger the `onReceive()` method with an empty SMS message. At line 5, the approach intercepts the first call to the environment, `getSMSBody()`, and queries multiple analyses for possible return values. Suppose that one analysis, which extracts string constants from the app's bytecode, suggests the value `"anserverb.db"`. Another analysis, which reasons about string operations by extracting and solving constraints, suggests a value `"#s:abc"`. Suppose that FuzzDroid randomly decides to return the value `"anserverb.db"`, so the `onReceive()` method returns without reaching the target.

Next, the approach re-executes the method and again reaches the fuzzing decision at line 5. Suppose that now, FuzzDroid fuzzes the `getSMSBody()` call by returning `"#s:abc"`, so the app takes the if branch and gets closer to the target location. During all executions, the approach keeps track of the distance of the executed path to the target location. FuzzDroid exploits this knowledge to prioritize values. For the example, the approach may infer from the first two executions that `"#s:abc"` leads the execution closer to the target, and is thus more likely to reach the target than `"anserverb.db"`.

To eventually reach the target, FuzzDroid proceeds to fuzz environment calls while improving the selection of fuzzed values, until it finds suitable return values for further environment calls at lines 6, 13, 30, 36 and 39. Key to reaching the target is to combine values extracted with several, complementary analyses, instead of relying on a single analysis. Note that we only consider a target location as reached if it can be executed without an exception. Sections 5.2 and 5.3 present our approach in more detail.

5.2 A TARGETED FUZZING FRAMEWORK

The FuzzDroid approach consists of a generic fuzzing framework and an extensible set of value providers for fuzzing particular APIs. This section presents the fuzzing framework. At first, we present a more detailed overview of the framework including the main components of FuzzDroid (Section 5.2.1). Afterwards, we explain the overall algorithm of the framework (Section 5.2.2). Then, we describe how the framework interacts with the app during the execution (Section 5.2.3). Next, Section 5.2.4 presents how the framework steers the execution toward a target location by picking appropriate values for the fuzzed APIs. Finally, we present how FuzzDroid deals with dynamically loaded code (Section 5.2.5).

The goal of FuzzDroid is to find an environment in which the app reaches a target location. An Android app interacts with its environment through API calls, such as `getDeviceId()` and `getSMSBody()`. We control the environment of an app by fixing the values returned by such API calls:

Definition 5 (Environment) *An environment* $\mathcal{E} : \mathcal{L} \times \mathbb{N} \to \mathcal{V}$ *is a map that assigns a value* $v \in \mathcal{V}$ *to a pair* (l, n)*, where* $l \in \mathcal{L}$ *is a code location and* $n \in \mathbb{N}$ *is a counter of how often* l *has been reached in the current execution.*

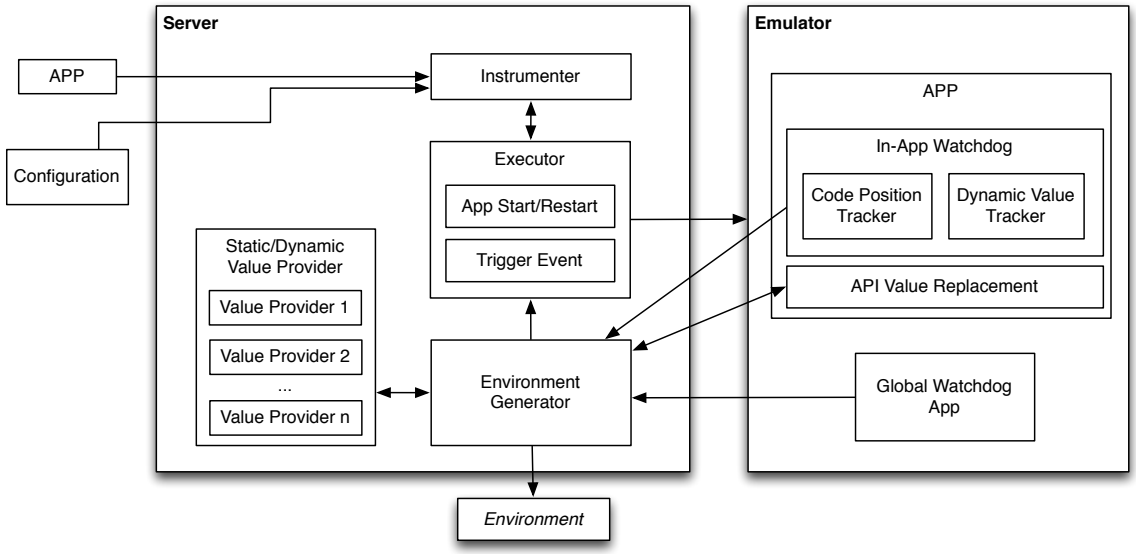

Figure 13: Detailed overview of the FuzzDroid approach

For example, suppose that l_{SMS} is a call site of `getSMSBody()`. The environment $\{(l_{SMS}, 1) \mapsto \text{"abc"}, (l_{SMS}, 2) \mapsto \text{"def"}\}$ specifies that the call to `getSMSBody()` returns `"abc"` and `"def"` when the location is reached for the first and second time, respectively. The counter n of the definition is mainly used by the value providers, which will be described in Section 5.3. For instance, the `File.readLine()` API is usually used in a while-loop. A value provider must thus first provide values for n=0,1,2..., but then $< null >$ at some point to break out of the loop for some n.

We call an environment that enables the app to reach a target location a *successful environment*.

5.2.1 *Framework Overview*

Figure 13 provides an overview of the main components of the FuzzDroid framework. In general, the framework consists of a *server* and an *emulator*. The server-part is responsible for extracting an environment in which the app reaches a target location and the emulator is used for executing the app and providing runtime information to the server. As shown in the figure, the framework takes as input an *app* and a *configuration file* that contains, among others, the target locations. The *instrumenter* enriches the app with additional functionality that is required for identifying the environment. It contains an *In-App Watchdog* that keeps track of the executed code locations and dynamic values. The latter one will be described in more detail in Section 5.3.1.3. Furthermore, it also contains a component that is responsible for replacing environment API values that are provided by the server. The *Global Watchdog App* is required due to implementation reasons, which will be explained in the implementation section (Section 5.4).

The extraction of the environment in which the app reaches a target location is extracted by the *Environment Generator*, which gets the necessary values from different *Static/Dynamic Value Providers*. A detailed description about the former component including different

Algorithm 1 Find an environment $\mathcal{E}$ that reaches location l_{target} in app.

Require: App app, set $\mathcal{L}_{target}$ of target locations, and set $\mathcal{A}_{fuzz}$ of APIs to fuzz
Ensure: Environment $\mathcal{E}$

```
1:  Q ← [app]                                              # app queue
2:  while Q ≠ empty do
3:    app_current ← Q.pop()
4:    staticPreAnalysis(app_current)
5:    instrument(app_current, L_target, A_fuzz)
6:    nbRuns ← 0
7:    T ← ∅                                                # trace pool
8:    while nbRuns < maxRuns do
9:      nbRuns ← nbRuns + 1
10:     E ← initializeEnvironment(T)
11:     trace ← executeAndFuzz(app_current, E, Q)
12:     T ← T ∪ {trace}
13:     if targetReached(trace, L_target) then
14:       report(E, trace)
15:       exit
16:     end if
17:   end while
18: end while
```

algorithms and information about the *Executor* is provided in the next section. Section 5.3 describes the individual value providers in detail.

5.2.2 *Main Algorithm*

In the following, we will focus on the *Environment Generator* and *Executor* components of Figure 13. To find a successful environment, FuzzDroid repeatedly executes the app while refining the environment, as summarized in Algorithm 1. The outer loop of the algorithm will be explained in Section 5.2.5; the reader should ignore it for now and focus on the steps starting from line 3.

At first, the algorithm triggers a static analysis of the app to build an inter-procedural control flow graph of the app, which will be used by the subsequent steps (usage of reflective method calls and dynamic code loading will be explained in Section 5.2.5). In addition, each value provider plugged into the framework can apply further static analysis at this point. Next, the framework instruments the app so that both can interact with each other when the app executes. Specifically, FuzzDroid adds instrumentation code to keep track of the execution path and to intercept calls to the fuzzed APIs (see Instrumenter in Figure 13).

The main loop of the algorithm starts at line 8. The framework repeatedly executes the app until either a target location or a configurable maximum number of executions has been reached. Before each execution, function initializeEnvironment creates an environment. Section 5.2.4 describes this step in detail. During the execution, the instrumented app queries the framework for values to be returned at call sites of fuzzed APIs. Section 5.2.3 presents the executeAndFuzz function, which implements this step, in detail. The framework summarizes each execution into an execution trace and maintains a pool of all previ-

ous executions. After each execution, the algorithm checks whether the target location has been reached (line 13). If the framework has reached the target, the algorithm returns the successful environment. Otherwise, the algorithm refines the environment based on the feedback obtained from previous executions and executes the app again.

5.2.3 *Executing and Fuzzing Apps*

The core of FuzzDroid's fuzzing happens in function executeAndFuzz, called at line 11 of Algorithm 1.

We will explain the individual parts of this algorithm based on the motivating example (Section 5.1). For the sake of simplicity, we assume that FuzzDroid needs to identify the correct environment for SMS-body (line 5), the certificate (line 6) and the mobile operator (line 13) for reaching the target location `sendTextMessage()` at line 45. More concrete, possible environment values are $\texttt{body} = \text{"\#s : abc"}, \texttt{certificateHash} = 389d90$ and $\texttt{mobileOp} = \text{"telecom"}$. In the following, we will use `body`, `certificateHash` and `mobileOp` for line 5, line 6 and line 13, respectively. The handling of the timing bomb (line 33) and contacts will be explained in Section 5.4 and dynamic code loading in Section 5.2.5.

5.2.3.1 *Fuzzing the Environment*

Each execution starts with an initial environment $\mathcal{E}$ that maps a subset of all possible API calls that may happen during the execution to return values. During the execution, the app queries the framework whenever the execution reaches a fuzzed API. If the app requires a pair $(l, n) \in \mathcal{E}$, i.e., a value provided by the initial environment, then the framework returns this value. Otherwise, the framework queries the value providers, selects one of the provided values, and adds this value to the environment $\mathcal{E}$.

To help the framework select a value, value providers associate with each value a weight that specifies the confidence the value provider has in the respective value. The weights also allow for prioritizing particular value providers over others, e.g., if one value provider is generally more precise than others. The framework selects a value by ordering all provided values and by picking randomly among the values with the highest weight. To prevent the framework from permanently rejecting values with low weight, the framework also considers all remaining values with a low probability (10% per default) and selects randomly among them, regardless of their weight. Algorithm 2 represents a possible implementation for the proposed value determination.

In addition to extending the current environment $\mathcal{E}$ with the selected value, the framework also keeps all other provided values in so-called *shadow environments*. These environments have not yet been used during an execution. The framework may pick from these shadows if no analysis is able to compute any further values for a given environment query (l, n).

During the execution of the app in the fuzzed environment, the framework summarizes the execution into a trace:

Definition 6 (Trace) *A trace* $t = (L, \mathcal{E})$ *summarizes the execution of an app into the list* $L = [l_1, ..l_n]$ *of executed code locations* $l_i \in \mathcal{L}$ *and the environment* $\mathcal{E}$ *that has triggered this execution.*

At the end of an execution, the framework adds the trace to a *trace pool*. These traces have two purposes. First, the framework creates future initial environments based on the traces

Algorithm 2 Determine value v for environment

Require: set $\mathcal{V}$ of values provided by value providers and a probability $p \in 1, 2, .., 99$ for picking non-high values
Ensure: value v for environment

```
1: V_H ← getAllValuesWithHighestScore(V)
2: V_R ← getAllValuesExceptHighestScore(V)
3: cmp ← 0
4: rand ← Random.nextInt(p)                    # random value between 0 and p
5: if cmp == rand then
6:    v ← pickRandom(V_R)                      # pick random value of set
7:    return v
8: else
9:    v ← pickRandom(V_H)                      # pick random value of set
10:   return v
11: end if
```

of previous executions, as described in detail in Section 5.2.4. Second, value providers can adjust the set of provided values based on the current trace and on the trace pool. For example, a value provider may reduce the weight based on values already used in previous executions or provide values based on the path taken in the current execution.

Let's assume for now that FuzzDroid run three times and produced the following trace pool

$$\mathcal{T} = (\left(\begin{pmatrix} \text{line 5,} \\ \text{line 6,} \\ \ldots, \\ \text{line 28} \end{pmatrix}, \begin{pmatrix} (\text{body, "\#s : abc"}), \\ (\text{certificate, 389d90}), \\ (\text{mobileOp, "aaaa"}) \end{pmatrix} \right), \left(\begin{pmatrix} \text{line 5,} \\ \text{line 6,} \\ \text{line 9} \end{pmatrix}, \begin{pmatrix} (\text{body, "aaaa"}), \\ (\text{certificate, 99999}) \end{pmatrix} \right), \left(\begin{pmatrix} \text{line 5,} \\ \text{line 6,} \\ \ldots, \\ \text{line 11} \end{pmatrix}, \begin{pmatrix} (\text{body, "\#s : abc"}), \\ (\text{certificate, 11111}) \end{pmatrix} \right))$$

Please note that we only show the environment and part of the executed code locations and we further omit the location-counter information for simplifying the formulas.

5.2.3.2 *Triggering Events and Services*

The approach described so far assumes that the target location is reachable by simply starting the app under a suitable environment. However, some target locations may only be reached when the app reacts to a particular event, such as an incoming SMS message or a click on a button. To enable the approach to reach such target locations, FuzzDroid programmatically triggers event handlers (see *Executor* in Figure 13). For this purpose, the framework computes a static callgraph of the app and traverses it backwards, starting

at the method of the target location. When reaching the beginning of a callback method, FuzzDroid checks the event for which the respective callback is registered and triggers the event programmatically. FuzzDroid directly calls the respective event handler method and thus does not need a full system model or a graphical UI model. Even if several UI interactions would be necessary to trigger the event during normal execution, it shortcuts all of them and directly jumps into the handler.

5.2.3.3 *Properties of Successful Environments*

Since FuzzDroid checks dynamically whether an environment reaches the target location, a reported environment is guaranteed to reach the target. In contrast, FuzzDroid guarantees neither to find a minimal environment nor to generate a realistic environment, i.e., a reported environment may over-constrain individual environment values, possibly with values that may not occur in reality. Suppose an app requires the name of the network operator to contain the string `"tele"` (e.g., telecom). In this case, FuzzDroid may report an environment that sets the network operator to `"teleFoo"`, which is not an actual network operator. Our evaluation shows that the absence of these guarantees is a non-issue in practice, because most successful environments specify a manageable number of values.

5.2.4 *Steering Towards the Target*

Since the set of possible environments that FuzzDroid can generate is too large to explore exhaustively, it is crucial to steer the approach toward an environment that reaches the target location. This section explains how FuzzDroid steers toward such an environment based on the trace pool. These steps of the approach correspond to function `initializeEnvironment` in Algorithm 1.

5.2.4.1 *Measuring the Fitness of Environments*

To identify environments that are likely to lead the app to the target location, we compute a fitness score for each environment based on the trace that the environment yields:

Definition 7 (Fitness of an environment) *Given a trace* $(L, \mathcal{E})$ *and a target location* l_{target}*, the fitness of* $\mathcal{E}$ *is the minimum distance between* l_{target} *and any location* l *in* L*.*

FuzzDroid computes the distance between code locations as the minimum number of edges between the locations in an inter-procedural control flow graph. The rationale for considering the minimum distance is that traces that get close to the target at some point during the execution are more likely to reach the target than traces that always remain far from the target. Applied to our running example,

$$F_1 = \left(\begin{pmatrix} \text{line 5}, \\ \text{line 6}, \\ \dots, \\ \text{line 28} \end{pmatrix}, \begin{pmatrix} (\text{body}, \text{"\#s : abc"}), \\ (\text{certificate}, 389d90), \\ (\text{mobileOp}, \text{"aaaa"}) \end{pmatrix} \right) = 10$$

since the minimal distance between the closest executed statement (the mobile operator check in line 28) and the target location is 10 statements. Correspondingly,

Algorithm 3 Create an initial environment $\mathcal{E}$

Require: Trace pool $\mathcal{T}$
Ensure: Environment $\mathcal{E}$
1: **if** $|\mathcal{T}| < \mathit{minTraces}$ or $\mathit{randomNb}() < 0.25$ **then**
2: **return** empty environment
3: **end if**
4: $E_{sorted} \leftarrow \mathit{sortEnvsByFitness}(\mathcal{T})$
5: **while** $E_{sorted} \neq$ empty **do**
6: $\mathcal{E}_1, \mathcal{E}_2 \leftarrow \mathit{selectParentEnvs}(E_{sorted})$
7: $\mathcal{E} \leftarrow \mathit{crossover}(\mathcal{E}_1, \mathcal{E}_2)$
8: **if** $\mathit{isNewEnv}(\mathcal{E}, \mathcal{T})$ **then**
9: $\mathit{mutate}(\mathcal{E})$
10: **return** $\mathcal{E}$
11: **else**
12: $\mathit{remove}(E_{sorted}, \mathcal{E}_1)$
13: **end if**
14: **end while**
15: **return** empty environment

$$F_2 = \left(\begin{pmatrix} \text{line 5,} \\ \text{line 6,} \\ \text{line 9} \end{pmatrix}, \begin{pmatrix} (\text{body, "aaaa"}), \\ (\text{certificate, 99999}) \end{pmatrix} \right) = 19$$

since the execution trace does not pass the SMS-body-check in line 9 and

$$F_3 = \left(\begin{pmatrix} \text{line 5,} \\ \text{line 6,} \\ \ldots, \\ \text{line 11} \end{pmatrix}, \begin{pmatrix} (\text{body, "\#s : abc"}), \\ (\text{certificate, 11111}) \end{pmatrix} \right) = 18$$

since the execution trace does not pass the certificate-check in line 11.
Other measures can be easily added to our approach.

5.2.4.2 *Evolutionary Algorithm*

Based on the fitness of environments, FuzzDroid influences the generation of environments using an evolutionary algorithm that creates new initial environments. The intuition behind the approach is that traces, which came close to the target are likely to have values in their environment that enable the app to reach the target. The algorithm repeatedly combines such successful environments into new environments, until FuzzDroid reaches the target.

Algorithm 3 summarizes the main steps. Given a trace pool $\mathcal{T}$, the algorithm computes an environment $\mathcal{E}$ for the next execution. At first, the algorithm checks whether the size of the trace pool exceeds a minimum value (five in our evaluation and two in our running example), and otherwise, returns an empty environment. The empty environment forces FuzzDroid to query value providers at runtime for values, which yields additional traces to learn from. If there are sufficiently many traces, the algorithm sorts the environments of these traces by their fitness score. Applied to our running example, the order of the

fitness scores F_1, F_2, F_3 would be (F_1, F_3, F_2). Next, the main loop of the algorithm (lines 5 to 14) performs the following classical steps of an evolutionary algorithm: select two parent environments, create a new environment through a crossover operation, and check whether this yields an environment that has not yet been used in any previous execution. If so, the algorithm mutates the new environment and returns it. Otherwise, the search continues until the algorithm runs out of possible parent environments. In this case, the algorithm returns an empty environment.

SELECTING PARENTS The *selectParentEnvs* function (line 6) selects the two environments with the highest fitness (F_1 and F_3 in our running example). If combining these environments yields an environment that has already been tried before, then the algorithm removes the current best environment from the sorted list E_{sorted} of environments to consider. Please note that the sorted list E_{sorted} from line 4 is only a copy of the original trace pool $\mathcal{T}$. Therefore, the removal of the environment does not influence $\mathcal{T}$, it is only applied on the sorted environment list. As a result, the next iteration of the algorithm's main loop selects the second-best and third-best environment as potential parents, and so on, until there are no more possible parent environments left.

Note that there can also be the opposite mode: If no analysis has generated any new values during the last dynamic execution, i.e., there was no other option than re-executing an already-seen environment, FUZZDROID switches to a genetic-only mode in which it always performs a genetic re-combination and never creates an empty environment.

CROSSOVER To combine two environments, FUZZDROID first computes the union of all keys in the environments. If a key is provided by only one environment, then this environment contributes the values. Otherwise, FUZZDROID randomly chooses which environment contributes the value. More concrete, the environments of F_1 and F_3 get combined as follows:

1. $(\mathtt{body}, \mathtt{"\#s : abc"})$ and $(\mathtt{body}, \mathtt{"\#s : abc"})$: $(\mathtt{body}, \mathtt{"\#s : abc"})$
2. $(\mathtt{certificate}, \mathtt{389d90})$ and $(\mathtt{certificate}, \mathtt{11111})$: FUZZDROID randomly collect one of both. Let's assume FUZZDROID takes $(\mathtt{certificate}, \mathtt{389d90})$
3. $(\mathtt{mobileOp}, \mathtt{"aaaa"})$: Since only F_1 contains this element, FUZZDROID's union step adds it to the new environment.

This results to a new environment $\mathcal{E}_{new} = ((\mathtt{body}, \mathtt{"\#s : abc"}), (\mathtt{certificate}, \mathtt{389d90}), (\mathtt{mobileOp}, \mathtt{"aaaa"}))$.

MUTATION To avoid getting stuck in a local minimum, i.e., an environment that brings the execution close to the target but based on which the execution cannot reach the target, the algorithm mutates the environment that results from crossover. This step allows the algorithm to explore new directions instead of only exploiting existing knowledge. Through mutation, "worse" environments that escape such minima can be evaluated. FUZZDROID mutates each value of an environment with a small probability (0.1 in our evaluation). To mutate a value for a particular API method, the approach picks from an environment, different from the two current parents, a random value provided for this API method. If no environment contains this value, FUZZDROID looks up a new value

from the *shadow environments* (see Section 5.2.3.1) or asks the value providers to provide a new value. Let's assume for our running example that FuzzDroid picked the $(\mathtt{mobileOp}, \text{"}\mathtt{aaaa}\text{"})$ in $\mathcal{E}_{new}$ for mutation. A lookup in the shadow environments resulted in the fact that "aaaa" get replaced to "telecom", which further results to a new environment $\mathcal{E}_{new} = ((\mathtt{body}, \text{"}\#s : \mathtt{abc}\text{"}), (\mathtt{certificate}, 389d90), (\mathtt{mobileOp}, \text{"}\mathtt{telecom}\text{"}))$.

In addition to mutation, the algorithm uses two other ways to evade local minima. First, the algorithm returns an empty environment with a configurable probability (0.25 in our evaluation) to ensure that FuzzDroid continuously tries new environments (line 2). Second, with a probability (0.1 in our evaluation), the $\mathtt{selectParentEnvs}$ function picks two random traces for combination instead of the two best ones.

FuzzDroid detects when an app execution crashes. In such a case, the environment that was executed at that time receives a penalty, i.e., a very low fitness value. This makes choosing environments that crash the app less likely than choosing any other environment, including those that fail to reach the target location.

5.2.5 *Dealing with Dynamic Code Loading*

Some malware apps hide malicious behavior by storing the malicious code in an encrypted file and by decrypting and loading this code at runtime. In this case, the malicious code is unavailable to the static part of our framework. In particular, the target location may not be visible to static analysis, making it impossible to, e.g., compute the distance between the target and already executed locations. FuzzDroid deals with such *packing* by observing dynamically loaded code and by rewriting the app into an app that contains this code. To this end, FuzzDroid first takes all locations of dynamic code loading as targets and attempts to steer the execution to these parts of the code. At runtime, the framework then obtains the dynamically loaded code right before it is being passed to the Dalvik/ART runtime. The code is then merged into the original APK file and future runs of the framework can operate on the full, un-obfuscated code.

Because an app may load different code on different execution paths, we do not simply overwrite the app but copy the current app and rewrite the copy. Algorithm 1 keeps track of the rewritten copies of the app in an app queue `Q`. We chose a queue as data structure since the first loaded dex-file should be merged first into the app, the second loaded dex file as second and so on. Initially, the queue contains only the original app (line 1). Whenever the framework observes dynamically loaded code, it copies the current app and adds the code to the copy. This is why the `executeAndFuzz` method is passed a reference to the queue `Q`. If, in a later execution of the original app, the framework detects other dynamically loaded code, the app will again be copied and have new code added to it, which does not interfere with the code added in a previous execution. We do not copy the traces, though, as new code can arbitrarily change the environment required to reach the target location.

To call methods from dynamically loaded `dex` files, apps use the Java reflection API. As a further obfuscation step, these method and class names are, not present as static strings. Instead, they are only computed or decrypted at runtime, making them unavailable to static analysis even if all parts of the code are available. To address this challenge, FuzzDroid applies HARVESTER (previous chapter) as a pre-analysis to each app. Harvester uses a combination of static and dynamic analysis to precisely extract the targets of reflective method calls and replaces them with direct calls.

5.2.6 *Determination of Target Locations*

To ease the definition of target locations, FuzzDroid provides a comprehensive list of predefined target locations, which is mainly taken from the SuSi approach (see Chapter 3). FuzzDroid makes use of the APIs within the sink categories. However, depending on the malware investigation, also the APIs within the source category need to be considered. Examples are the access to the `getInstalledApplications()` API for instance, which can be used for benign, but also in malicious purposes (more details provided in Section 6.2). We manually add APIs for dynamic code loading and reflective method calls. FuzzDroid also supports the manual adding of code locations that are of interest for the human analyst.

5.3 VALUE PROVIDERS

This section presents a set of value providers, which create values to be returned at call sites of fuzzed APIs.

5.3.1 *Symbolic Value Provider*

Reaching the target location typically requires the app to take a particular path, and taking this path requires particular environment values. The way an app uses the values obtained from the environment often reveals semantic information about the expectations that the app has on environment values. For example, line 9 in Listing 11 reveals that the incoming message must start with `"#s:"`. Likewise, a call to `sendTextMessage(nr, body)` reveals that `nr` is expected to be a number or to start with a `"+"`, e.g., `"+491234"` or a call of `String.substring(0, 5)` reveals that a string should contain at least six characters.

To exploit such information, FuzzDroid contains a constraint-based, symbolic analysis that reasons about the usages of environment values in the app. The basic idea is to encode the results of a local, static analysis and values extracted at runtime into constraints, and to query a constraint solver to find values for fuzzing. The analysis computes for each call site $\mathcal{L}_{\texttt{fuzz}}$ of a fuzzed API a set of values. The approach consists of three steps explained in the following.

5.3.1.1 *Static Dataflow Analysis*

The approach statically reasons about the uses of fuzzed values through an inter-procedural dataflow analysis. Given a set of source locations and a set of sink locations, the analysis extracts sequences of statements that propagate and modify values between a source and a sink. We call such a sequence of statements a *dataflow path*. The sources for our analysis are all call sites of fuzzed APIs. As sinks, we consider all call sites of a configurable set of methods that reveal expectations by the app on value, such as `String.startsWith()` and `sendTextMessage()`. The static analysis yields a map $\mathcal{L}_{\texttt{fuzz}} \rightarrow \mathcal{D}$ from call sites of fuzzed APIs to sets of dataflow paths.

For example, Listing 11 yields a dataflow path that connects `getSMSBody()` (line 5) with `sendTextMessage()` (line 45) via `startsWith("#s:")` (line 9) and `spamMessage` (line 29).

5.3.1.2 *Constraint Solving*

The analysis translates the dataflow paths into constraints and solves them to obtain values to be suggested at call sites of fuzzed APIs. At first, the approach transforms each dataflow path into a static single assignment (SSA)-form [AWZ88], where each variable is assigned at most once. This step is necessary since our implementation is based on the Jimple [ARB13] intermediate representation that re-uses variable names. This can result to the case that a dataflow path contains two or more different statements where the left-side of an assignment has the same variable name. Next, the approach translates dataflow paths into conjuncts of constraints understood by the Z3 solver [DMB08], similar to prior work [Arz+15]. Specifically:

- The initial call of a fuzzed API method in each dataflow path is represented as a symbolic variable $v_{\texttt{fuzz}}$.
- Any operations applied in the dataflow path are translated into the corresponding Z3 constraints. For example, we translate string operations into their corresponding constraints provided by the Z3 string theory [ZZG13]. For API methods not supported by the Z3 solver, such as `String.split()`, we provide additional constraints that model the behavior of the respective API.
- To encode the information revealed by the call of the sink method at the end of each dataflow path, we translate this call into appropriate constraints. For example, a call to `sendTextMessage()` is encoded as constraints that specify that the first argument provided to the method must be a number or "+" followed by a number.
- For dataflow paths with sink methods that represent Boolean statements, such as `String.equals()`, we generate two conjuncts of constraints, which represent the case that the check returns *true* and *false*, respectively. Our experiments show that this, albeit somewhat crude, approach is highly effective when applied to current malware. Furthermore, it is much faster as if we would have added additional symbolic evaluation for detecting the correct path. Please note that our approach is based on fuzzing, which does not produce any issues if both conditions are considered.

This translation yields a map $\mathcal{L}_{\texttt{fuzz}} \to \mathcal{C}$ from call sites of fuzzed APIs to sets of symbolic constraints. To compute values for fuzzing, the approach queries the solver for each set of constraints to obtain a concrete value for $v_{\texttt{fuzz}}$. The solving yields a map $\mathcal{L}_{\texttt{fuzz}} \to \mathcal{V}$ that assigns to each fuzzed location a set of possible values. Whenever the framework queries the analysis for a value to be returned at a location $l \in \mathcal{L}_{\texttt{fuzz}}$, the analysis returns one of the possible values. To reduce the computational cost of constraint solving, the analysis solves all statically extracted constraints before executing the app for the first time.

5.3.1.3 *Dynamic Refinement of Constraints*

The statically extracted constraints may contain symbolic variables in addition to the fuzzed value $v_{\texttt{fuzz}}$. For example, suppose an app compares the return value of a fuzzed API to a dynamically created string using `String.equals()`. Without knowing the dynamically created string, the constraint solver is unlikely to return a suitable value for $v_{\texttt{fuzz}}$, because it knows only that $v_{\texttt{fuzz}}$ is equal to another symbolic variable $v_{\texttt{unknown}}$. We address this problem by enriching the statically computed constraints with dynamically extracted values. To this end, the analysis obtains from the framework runtime values involved in calls

of a configurable set of methods. By default, we include into this set of methods string operations, such as `String.equals()` and `String.substring()`, because these operations are particularly important in various malware apps. If an execution produces a concrete value c for a symbolic variable $v_{unknown}$ in one of the statically computed constraints, then the approach copies the constraint and, in the copy, replaces the $v_{unknown}$ with c. By solving the refined constraints, the analysis is more likely to obtain a suitable value for v_{fuzz}. For the above example, the refined constraints specify that v_{fuzz} is equal to c, making it trivial to find a precise solution.

To reduce the cost of constraint solving, the analysis performs the dynamic refinement of constraints on demand. That is, whenever the framework queries the analysis for a value at a location, the analysis checks whether the constraints for this location contain any symbolic variable for which concrete runtime values have been observed in previous executions. Only if such runtime values exist, the analysis gives the refined constraints to the solver.

As an example, please consider line 13 till line 28 from the motivating example in Section 5.1. A constraint for *mobileOp* would be $v_{mobileOp} = v_{line}$ where v_{line} is not known during the static phase since it gets assigned with dynamic values. However, our *Dynamic Value Tracker* shown in Figure 13 sends all dynamic values of the `String.equals()` (line 22) method to the value providers. Therefore, in the next run, the *Symbolic Value Provider* refines its constraint to

$$v_{mobileOp} = \text{"telecom"} \vee v_{mobileOp} = \text{"megafon"} \vee v_{mobileOp} = \text{"mts"}$$

if "telecom", "megafon" and "mts" is included in the encrypted file (line 21).

Since the results computed from constraints that make use of dynamic values are usually more precise than those that rely only on static dataflow paths, the approach gives them a higher weight when providing them to the framework.

Our constraint-based analysis differs from traditional symbolic and concolic execution [Kin76; GKS05; SMA05; GLM08; CDE08b; Cad+08; Sax+10] by applying a local symbolic analysis instead of reasoning about the entire execution path. The benefit of avoiding path sensitivity is that our local analysis scales well to large apps. However, the approach cannot guarantee that values obtained from the solver will cause the app to reach the target. For example, our analysis does not reason about which of the two branches of a conditional leads to the target, but instead, suggests values for both branches. Fortunately, since FuzzDroid is a fuzzing approach that validates by executing the app whether an environment reaches the target, the symbolic analysis need not to provide this guarantee. Instead, FuzzDroid iteratively selects suitable values based on the fitness of executed environments (Section 5.2.4).

5.3.2 *Constant Value Provider*

Many apps compare runtime values against constants stored in the code. These constants may not be directly in a condition statement but, e.g., read from variables or fields. To execute branches guarded by such conditionals, we have implemented an analysis that gathers from an app's bytecode all constants of primitive types and strings. The value provider returns these constants when being queried for a value of a matching type. If the app does not contain any statically extractable constants, e.g., due to obfuscation, the analysis returns values from a pre-defined pool of random values.

For the example in Listing 11, the constant value provider helps pass by the integrity check at line 11. The value provider extracts the hash value of the certificate at line 2 and suggests it when the app queries the certificate hash at line 6.

5.3.3 *File Value Provider*

For some malware apps, the existence of a file, possibly containing data of a particular file type, is essential for triggering malicious behavior [Ras+15b]. To prevent apps from failing when an expected file is missing, the file value provider suggests values for APIs that access the file system. If the accessed file does not exist, the provider emulates the file. Such scenarios usually occur if the app itself, or some other app installed on the same phone, creates the file under specific circumstances.

Since some apps crash when files do not contain data of the expected type, the file value provider infers the expected file type and provides a dummy file of the inferred type. During the static pre-analysis, the analysis approximates the set of possible file types for this particular file using a forward dataflow analysis. The idea is to follow the dataflow from the file access to an API call that reveals the expected file type. For example, a dataflow that reaches a `SQLiteDatabase.openOrCreateDatabase()` call reveals that the file is expected to be a database file. We provide a manually assembled map between API calls and file types. Once the type is known, a manually created dummy file of the correct type is picked and pushed onto the phone before the app accesses that file. If the analysis fails to statically identify the expected file format, it tries to create a suitable file based on the name of the accessed file.

Note that this analysis is not responsible for creating a file with correct content. The file must only be in a shape that allows for successfully loading it. Further API calls that, e.g., read data from the file are intercepted separately. Other analyses such as the symbolic value provider (Section 5.3.1) can then provide the values expected to be read from the file.

5.3.4 *Value Provider for Integrity Checks*

Many malicious apps protect against code modifications by validating their own integrity through a check of the app's certificate, which is used for signing an Android app. When modifying the app's bytecode, one must re-sign the app. However, without access to the original developer's private key, it is practically impossible to use the same certificate for signing the app. A common way to implement an integrity check is to compute the hash code of the signature certificate extracted from Android's package manager and to compare it to an expected value. Alternatively, an app may also use the computed hash to decrypt data, such as additional, dynamically loaded code.

To circumvent such checks, FuzzDroid contains a value provider that fuzzes API calls that access an app's signature. Instead of the real certificate of the actually running app, the value provider returns the certificate that was used to sign the original, un-instrumented app, effectively fooling the check.

5.3.5 *Primitives-as-Strings Value Provider*

Java supports various API calls for data type conversion, e.g., to convert a string into a numeric value type. These methods are usually called `valueOf()` or `parse()` and give additional hints on the expected format of unknown data. Assume data is read from a file as a string. If the data is later on converted to an integer, this constrains the possible contents of the file to be emulated. The most common case are Boolean flags. Many malware apps expect files such as Android's shared preferences to contain strings like `"true"` or `"false"`, which are then converted to Boolean flags at runtime and which define whether the malicious code behavior gets executed or not. Most of the time, the behavior is disabled by default. Only upon, e.g., a command from a remote command-and-control server, the app enables the flag in the file, i.e., changes the file contents to `"true"` to enable the malicious behavior.

The primitives-as-strings value provider uses static dataflow analysis to track dataflows from file code locations where dynamic values are obtained (e.g., file reads or the `Properties.getProperty()` method) to data type conversion methods. If the analysis is able to identify the type of the primitive, the value provider picks a value from a pre-defined set of random values of the correct type.

5.4 IMPLEMENTATION

We use Soot [VR+99] with the Dexpler front-end for Android [Bar+12] to statically analyze and dynamically instrument the apps. For implementing dataflow analyses, we build upon FlowDroid [Arz+14b]. The FuzzDroid framework runs on a desktop computer or server while the app runs on an Android emulator. Both communicate via a TCP connection, e.g., to request environment values or to report which path is executed (*Code Position Tracker* in Figure 13). To intercept fuzzed API calls, we use a user-space hooking library (ZHookLib[1], based on the Xposed framework[2]), which is represented as *API Value Replacement* in Figure 13. For most requests, the hooking component queries the server using TCP and directly injects the server's response into the app. This is, however, not suitable for transferring large objects such as dynamically loaded `dex` files, because the Android operating system enforces a strict time limit on its callbacks. If a dex file is loaded (and thus sent to the server) in a callback, Android is likely to kill the app due to a timeout. We therefore implemented a secondary, one-way communication mechanism that stores data on the emulator's SD card. A separate *Global Watchdog App* (see Figure 13) that runs in parallel on the same emulator monitors this folder, picks up all files stored there, and sends them to the server for analysis. Details about FuzzDroid's handling of dynamic code loading is provided in Section 5.2.5.

A similar performance problem exists for methods that are frequently called. While the hooking library only intercepts calls within the app's process and not system-wide, this still includes many internal calls that happen inside the usermode part of the Android framework. Though we are not interested in those calls, the fact that they are hooked, delays every call slightly. In sum, these delays can add up and cause a timeout. We therefore handle such "hot" methods separately through instrumentation instead of hooking which allows us to a priori pick only the code locations of interest. In contrast, using a global

1 https://github.com/cmzy/ZHookLib
2 http://repo.xposed.info/

instrumentation would be more complex, since every method and class in Android has a limited amount of statements. It is very likely that this limit gets exceeded if we apply a global instrumentation instead of API hooking. On the other hand, fine-tuning the API-hooking that it disregards the Android framework was not possible with the ZHookLib library.

Some malware apps contain timing bombs, i.e., actions that are only executed after a certain time. To avoid having to wait for this time span, we statically patch those statements during our instrumentation phase and decrease the waiting time to a few seconds. These timing-based API calls are pre-defined by us and can be easily extended. Additional instrumented code notifies FuzzDroid that such a location was reached at runtime.

There are applications that require the existence of different contacts on the smartphone, e.g., used for spam distribution among the contact list. Since every default emulator does not contain contacts on the smartphone, FuzzDroid adds different contacts to the smartphone after it installed a new app on the emulator.

5.5 EVALUATION

We evaluate the effectiveness and efficiency of FuzzDroid by applying it to 209 malicious Android apps. Our evaluation focuses on the following research questions:

- How effective is the approach at finding an environment that enables an app to reach a target location?
- How effective are FuzzDroid's use of multiple analyses and the way these analyses are combined with each other?
- How efficient is the approach?
- What do the environments generated by the approach reveal about real-world malware?
- How does FuzzDroid compare to the best existing approach for generating inputs that steer an Android app toward a particular location?

5.5.1 *Experimental Setup*

We randomly collected 300 recent malware apps from VirusTotal[3] in June 2016. As target locations, we use call sites of seven API methods related to SMS messages. Sending SMS messages and aborting incoming SMS messages are a common threat for Android applications [Nak14; Anda]. Based on a manual pre-analysis, we found that the interception of SMS messages and the sending of SMS messages often require complex environments. Other interesting target locations for malicious applications such as Internet connections are most of the time immediately executed without any specific environment setup. This is the reason why we focus on SMS related API methods. Out of the 300 apps, 209 apps contain at least one target location. As environment APIs, we use 32 different API methods. We run FuzzDroid on a server with 64 Intel Xeon E5-4650 CPUs running at 2.70 GHz and 1 TB of physical memory. We configured FuzzDroid with at most 15 executions and to start the genetic recombination after 5 runs.

3 Online malware database: `https://www.virustotal.com/`

	Approach		
	Launch	Launch & trigger	FuzzDroid
Apps with $\geqslant 1$ target reached	31.28%	45.02%	75.12%
Target locations reached	10.48%	15.82%	62.34%
When target reached (min/avg/max):			
- Executions	1/1/1	1/1.33/6	1/3.20/14
- Time to target (seconds)	3/8/18	3/19/310	3/62/1469
- Size of environment	—	—	0/3.69/47
- Contributing analyses	—	—	0/1.14/4

Table 15: Overview of results. For values summarized over multiple applications, we provide the minimum/average/maximum values.

5.5.2 *Effectiveness in Reaching a Target Location*

We evaluate the effectiveness of FuzzDroid at finding an environment where the application reaches a target location. Furthermore, we compare the approach to two simpler approaches: 1) simply launch the app and hope that it will reach the target location without further intervention, and 2) launch the app and trigger specific events, such as clicking a button or sending an SMS message, as described in Section 5.2.3.2. In neither of the two simpler approaches we generate a particular environment. Instead, if the app calls environment APIs, the emulator's default values are returned.

Table 15 shows our results. We find that running the app under the "right" environment is crucial for reaching the target location. The default environment of the emulator is insufficient for most current malware. Furthermore, the results show that FuzzDroid is effective in generating an environment that successfully reaches the target location. In total, the approach reaches 240 different target locations (62.34%), which are part of 75,12% of all apps. Line 1 in Table 15 contains all those cases where at least one logging point is part of an application, while line 2 contains all logging points (there can be more than one logging point within an application).

In some cases, FuzzDroid fails to reach the target location. For example, several malware apps contain malicious behavior, which is however, not yet enabled, and thus never called. We conjecture that the app will be updated at some point to actually activate the malicious behavior that is already present in the code. Other reasons for not reaching a target location is covered in our limitations section (see Section 5.6) or is due to our current handling of initial triggers (see Section 5.2.3.2).

5.5.3 *Importance of Multi-Analysis Approach*

To evaluate how much the individual analyses of FuzzDroid contribute to the framework's overall effectiveness, we first evaluate the framework with all analyses enabled. Then, we disable each analysis in turn, i.e., run the framework with all but one analysis and then run it with a single analysis. The result indicates how much the effectiveness decreases if this analysis is left out. The results in Figure 14 show that there is no single analysis that can be left out without a negative impact, i.e., all analyses are of value. This shows that

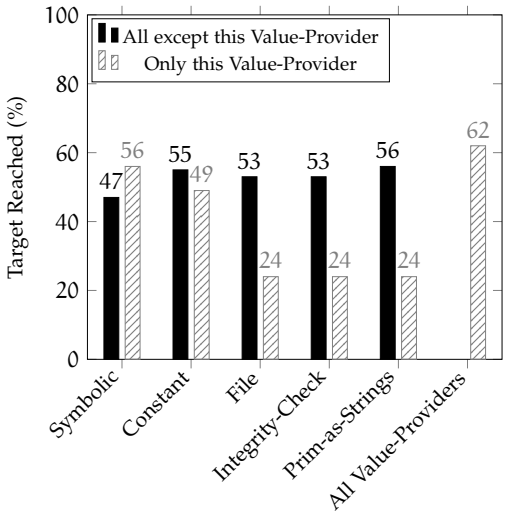

Figure 14: Comparison of effectiveness in reaching target locations for different subsets of all value providers

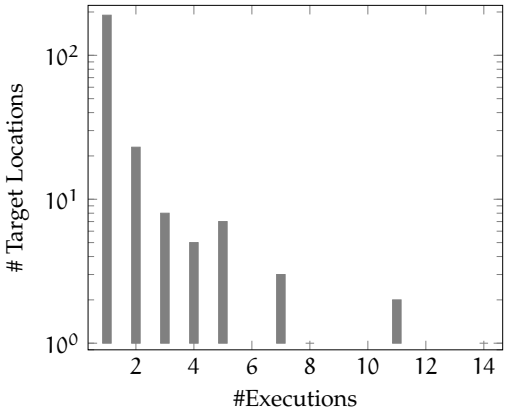

Figure 15: Amount of executions for reaching a target location

only a framework supporting multiple interacting analyses such as FUZZDROID is able to find correct execution environments for state-of-the-art malware.

We also found that overly simplistic analyses can decrease the effectiveness of the framework, i.e., the percentage of target locations reached. Choosing only random values, for instance, often leads to semantically invalid values. If the app reads the phone's IMEI, it can reasonably expect it to contain only digits. While the constant analysis will return digit- and non-digit strings, the probability of getting a digit-string from a purely random analysis is much lower.

5.5.4 *Efficiency*

Table 15 shows how long FUZZDROID takes to find an environment under which the application reaches the target location. On average, it takes 62 seconds to reach a target location. Most of the time (75%, 46.5 seconds) is spent for (repeatedly) executing the app. In contrast, both instrumenting apps (5%, 3.1 seconds) and statically analyzing apps (20%, 12.4 seconds) are secondary for the overall time.

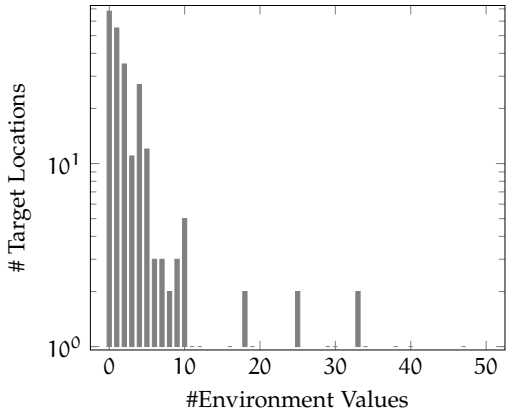

Figure 16: Amount of environment values for reaching target locations

Figure 15 shows the number of executions required by FuzzDroid to reach the target location. In most cases, the target location is reached in a single execution, i.e., the first generated environment is sufficient. For a substantial number of apps, however, the approach tries two to seven environments, showing that incremental fuzzing is required to reach the target location. The maximum number of execution needed during our evaluation is 14.

5.5.5 *Environments Generated by* FuzzDroid

5.5.5.1 *Number and Size of Environments*

The environments required to reach a target location range from trivial environments, where simply starting the app in the default emulator is sufficient, to complex environments with dozens of values. For the 75.12% of all apps where a trivial environment is insufficient, reaching the target locations requires 106 different environments. Manually creating these environments would be impractical for a human analyst, showing the need for approaches such as FuzzDroid.

Figure 16 shows the sizes of the environments generated by FuzzDroid. For several target locations in malware apps, no particular environment is required, i.e., the environment size is zero. For many others, two to ten different environment values must be combined to reach the target location, showing that FuzzDroid is highly beneficial for a security analyst interested in triggering malicious behavior. In some cases, the environment even consists of more than 30 values.

5.5.5.2 *Examples of Environments*

Beyond being useful for security analysts, FuzzDroid allows us to better understand how current malware interacts with the Android environment. The environments generated for several apps show that *targeted attacks* against a particular country, network operator, etc. are common in current malware. Table 16 summarizes which kinds of values we find in the environments that reach a target location. The following discusses several representative examples.

A very common kind of interaction with the environment is to access information from files. Besides such file accesses, various malware apps target particular SIM/network operators or check the SIM country code. For example, some apps expect SIM operator names to be "mts" or "megafon", two prominent network providers in Russia, or to match the

Kind of environment values	Prevalence
File access	47.97%
SIM/network operator code	16.82%
Specific incoming SMS message	10.84%
SIM operator name	5.53%
Timing bomb	4.06%
SIM country	3.16%
Integrity check	1.02%
Admin check	0.68%
Others	9.92%

Table 16: Prevalence of different kinds of environment values

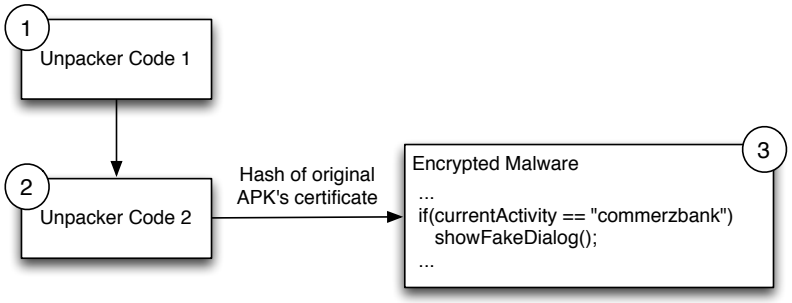

Figure 17: Workflow of packed Android malware

regular expression "*tele*", as in "telecom". We also find several malware apps that target specific countries, either by attacking users in a particular country or by checking that users are not located in a particular country.

Another interesting interaction with the environment are so called *timing-bombs*, where the malicious behavior only gets executed after a specified time has passed. This technique has been crucial for many malware samples to be accepted into the official Google Play Store without being detected [Coo+09]. Another technique popular among malware writers are integrity checks implemented to thwart bytecode instrumentation. Perhaps surprisingly, relatively few of the apps in our sample check whether the user grants the app device administration privileges.

Packed Malware

During the evaluation we encountered a malware app[4] with unusually few classes. Since all API calls are obfuscated through reflection, the malicious behavior is not directly visible. Instead, the app uses two additional files to hide more code. Figure 17 gives an overview over the app's workflow. In total, there are 3 different dex-files involved (highlighted as a circle). The *Unpacker Code 1* contains a byte-array that further contains a new dex-file (*Unpacker Code 2*), which gets dynamically loaded and which is responsible for unpacking the encrypted malware. The original malware is encrypted with the hash of the certificate of the application. Therefore, the *Unpacker Code 2* extracts the hash to unpack the original

4 MD5: 012ef9403fc221c84d7f9b43d51869c0

malware (*Encrypted Malware*). The dependency between certificate and decryption key is supposed to protect the integrity of the malware against, e.g., bytecode instrumentation. If the original APK file is modified, e.g., through bytecode instrumentation, the decryption fails unless the correct original certificate hash is injected. All these techniques are part of the commercial DexProtector [LY16] packer, which was applied on the malicious app.

After discovering through manual inspection that the malware may show a fake user interface asking about user name and PIN, we set the `showFakeDialog()` call as the target location. FuzzDroid then finds an environment that circumvents the integrity check and makes the malware app believe that the app of "Commerzbank" (a major bank) is opened. Only if both conditions hold, the phishing dialog prompts the user to her credentials as shown in Figure 18.

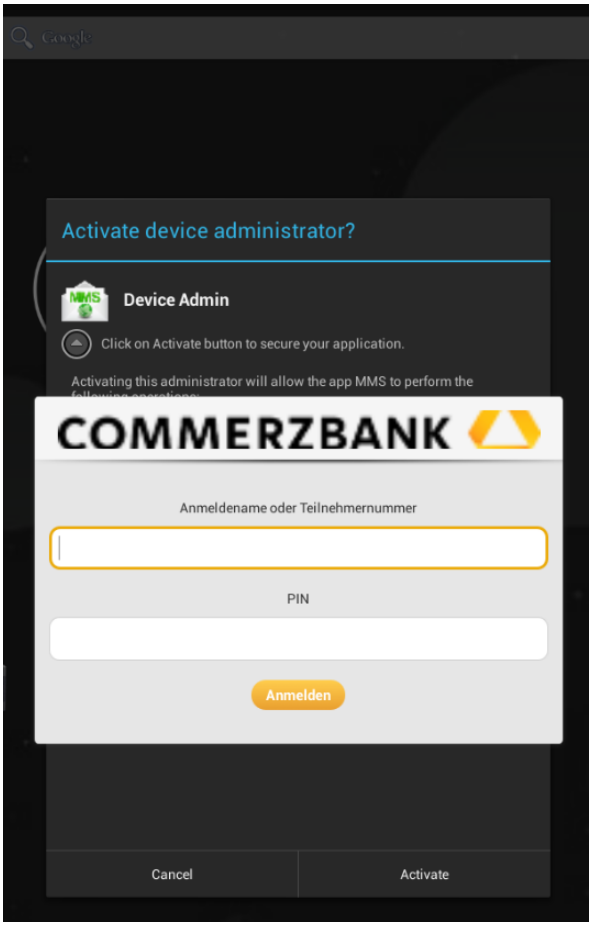

Figure 18: Phishing dialog for stealing Commerzbank credentials

This example illustrates (i) that FuzzDroid is able to handle applications with dynamically loaded dex-files (two in this case), (ii) what kind of techniques current malware uses for hiding its malicious behavior.

5.5.6 *Comparison with State-of-the-Art Approach*

IntelliDroid [WL16] is the conceptually closest approach to FuzzDroid. Yet, there are important differences. First, IntelliDroid is solely based on constraint solving and does not support multiple analyses. Second, IntelliDroid has a different notion of runtime values.

While we consider all values that are dynamically computed inside the app as runtime values, IntelliDroid obtains only the current device state, such as the current time or the registered alarm managers. Only this information becomes part of the constraint system. Therefore, if an app (such as the one in Listing 11) dynamically decrypts data, this data is unavailable to IntelliDroid. Third, IntelliDroid does not intercept API calls made within the app. It only relies on externally triggered events, such as sending an SMS message. This approach fails for the check on the mobile operator in our example. Fourth, IntelliDroid runs the app under analysis only once and therefore cannot obtain additional knowledge during this run to revise the constraint system and try again. Finally, IntelliDroid requires a lot of manual interaction with the tool and provides little automation.

Due to the lack of automation, we apply IntelliDroid to a random sample of 20 of our malware apps. We chose apps that contain at least one target location that is not immediately reached when starting the app or when triggering an event. In total, in this sample IntelliDroid reaches 11% of the target locations, whereas FuzzDroid reaches 62%. We conclude that FuzzDroid successfully addresses important limitations of IntelliDroid that prevent the state-of-the-art tool from reaching various target locations.

5.6 LIMITATIONS AND SECURITY ANALYSIS

Multi-Path Triggers

FuzzDroid currently assumes that it is sufficient to trigger a single path to reach the target location. If an app, on the other hand, requires a first SMS message to set a flag and then only executes the malicious code when a second SMS message is received after the flag has been set, FuzzDroid cannot trigger the malicious behavior. This is because we do not model the dependency between the two events and the flag is not considered as an environment value for FuzzDroid. Such dependencies are subject to future work. However, if the flag gets compared with an environment value at some point in the code, FuzzDroid might be able to extract the correct environment, e.g. with the help of the dynamic value provider. This highly depends on the implementation of such checks.

Callgraph Construction

Our current research prototype is based on an intra-component callgraph generated with the help of FlowDroid [Arz+14b]. To detect environment checks that are distributed across multiple components, FuzzDroid would need to be extended by an inter-component callgraph as described in Section 4.4.2. This is subject for future work.

Incomplete Implementation Support

An attacker who wants to circumvent FuzzDroid can try to attack the implementation by exploiting the lack of support for specific features. For instance, the current implementation does not support all possible environment APIs. As an example, it does not support Internet communication APIs, e.g., HTTP get/post requests. This means command-and-control messages received and send from and to a malicious server are not evaluated by now. The extension of FuzzDroid is subject to future work.

5.7 RELATED WORK

Various approaches for analyzing environment-dependent code are already known to literature. The topic has been of interest for revealing malicious behavior in suspicious apps as well as for increasing the test coverage of benign apps. Both aspects are related, as full code coverage would trigger the behavior of interest as well, though not specially targeted at the respective code position.

MALWARE ANALYSIS In *X-Force*, Peng et al. [Pen+14] propose a binary analysis engine which forces the execution of a program into specific branches. Their approach makes the program agnostic of the execution environment, revealing hidden behavior in malware. GoldenEye [Xu+14] exploits several virtual environments executed in parallel. The tool adaptively switches the analysis environment at runtime through a specially designed speculative execution engine. Moser et al. [MKK07] addresses a similar problem for x86 code as we do in our work for Android. His tool uses a dynamic approach in combination with system snapshots of the execution to execute code statements that produce a malicious behavior. To explore different paths, the program state is reset to earlier snapshots. The values on which the conditionals depend are updated to force different branches when the execution is resumed. This approach, however, requires linear relationships between all variables in the program to ensure consistent updating. Our approach restarts the program and steers the execution from the beginning to avoid this issue. Kolbitsch [Kol+12b] proposes a hybrid approach that combines a light-weight form of static symbolic execution with an instrumentation of additional code statements for a multi-path execution on JavaScript code. Abraham et al. [Abr+15] also propose a hybrid approach for reaching a certain target location. Their approach has an success rate of 28% and less on current Android malware and does not report information about the environment. TriggerScope [Fra+16] is a pure static approach that relies on symbolic execution for extracting environment information. The drawback of this approach is the static part, which comes with a lot of limitations, which prohibits the full extraction of the environment in many current malware samples. FUZZDROID in contrast is a hybrid approach and the extracted environments are verified to be correct by the dynamic part of our approach. IntelliDroid [WL16] uses constraint solving for generating environments under which a target location is reached. In contract to our approach, the tool runs the app a single time and has no feedback loop.

TEST CASE GENERATION Thummalapenta et al. [Thu+11] propose a mechanism for increasing test coverage by generating proper method call sequences. They first run the program to gather runtime information, which is then used to statically reason about non-executed branches. The execution is afterwards steered into those missing branches. ShamDroid [Bru+15] uses constraint solving to generate app-specific mocks for environment data to not give away real data during app execution while retaining the app's functionality. DART [GKS05] uses directed automated random testing to improve test coverage. The program is first executed using random inputs. Afterwards, symbolic execution is used to generate new inputs that steer the execution into paths that have not been explored yet. This way, it already improves over classical symbolic testing such as KLEE [CDE08a] or EXE [Cad+08]. Mirzaei et al. [Mir+12] create an Android system model in Java Pathfinder [HP00] to apply symbolic execution to the whole app for increasing test coverage. Similar to our approach, Malburg and Fraser [MF11] combine symbolic execu-

tion based on Java Pathfinder with a genetic algorithm that negates individual conditions during mutation. EvoDroid [MMM14] focuses on promoting the genetic makeup of good individuals during the genetic recombination in evolutionary testing for Android apps. Jensen et al. [JPM13b] use concolic execution to first summarize the effect of event handlers on the program state. Then, backwards from the target location, it composes these event handler summaries to find a path that reaches a given target.

5.8 SUMMARY AND CONCLUSION

In this chapter, we showed that reaching a certain target location in applications that contain different anti-static and anti-dynamic obfuscation techniques is possible if one applies a fuzzing based approach that makes use of different static and dynamic code analysis approaches. Pure static or pure dynamic approaches have too many limitations to be able to solve this problem. This is the final major contribution of this dissertation and answers the fundamental research question in the beginning of this chapter.

6

THE CODEINSPECT BYTECODE ANALYSIS TOOL

App store operators usually check thousands of apps per day against developer policy violations [Anda]. This includes, among other checks, checks against malicious applications that are uploaded by the developer. They usually have some form of automated code review process to handle this large amount of apps. For example, Google's application review process is based on a machine-learning approach that makes use of static and dynamic code analysis approaches [Anda]. However, a fully-automated app review process that is able to identify all possible malicious applications is not feasible in practice as explained by Google [Anda]. Limitations of the automatic code analysis approaches [Ras+15b] hinders most of the time a fully-automated detection. This gets exploited by attackers in implementing applications that circumvent the detection process [Nak14]. Google's automated review process includes manual reviews [Anda]. A human has to manually verify an application in cases where the machine-learning approach is not completely able to judge about potential policy violations. In such cases, human analysts reverse engineer the application's code and check whether the detection of the machine-learning approach was correct or not. If the human decides that the application does not violate the developer content policy, the application will be published in the Play Store.

Other examples that require manual code inspection through reverse engineering are malware analysts at antivirus companies who need to create virus signatures [Szo05] for malicious applications. This is most of the time a manual process and therefore very time consuming. Based on a personal interview with a malware analyst, she has on average about 30 minutes per application to decide whether the application is malicious or not. This shows that there is a need for good reverse engineering tools, for a faster analysis of an application.

There are different open-source [Sma; Jad; Jdg; Apk] and commercial [Jeb; Ida] solutions for manual reverse engineering Android applications. Most of them are limited to a static inspection of the app's bytecode, an intermediate representation of it or a decompiled Java-version. In the latter case, there are many different obfuscation techniques [BH07] that make it very hard, if not impossible, to convert an Android binary application into its original Java source code. Therefore, many human analysts inspect the bytecode on an intermediate representation level. Some of these approaches further include debugging functionality, which allows the analyst to step through the code during execution. However, due to the lack of type information in their intermediate representations, it is in many cases not easy to debug such an Android application.

Therefore, we introduce a new Android reverse engineering tool called CodeInspect. One of the main goals of this tool is to help human analyst in speeding up their manual reverse engineering task. This is achieved by different features, including the three approaches that have been described in the previous chapters. SuSi (Chapter 3) provides insights about sensitive API calls in an application, HARVESTER (Chapter 4) is able to extract runtime values even from obfuscated applications and is able to de-obfuscate reflective method calls and FuzzDroid offers the analyst insights about the concrete environment in which an application has to run for reaching certain code positions.

The main purpose of this chapter is to show how SuSi, HARVESTER and FuzzDroid can be integrated into a tool that can be used during a malware investigation. Furthermore, we want to show that the combinations of the proposed approaches, including other features of CodeInspect (will be introduced in the following), provide a lot of insights of the behavior of an application. We will demonstrate the tool interaction on a real malware investigation, which was introduced in the beginning of this thesis (Chapter 1).

Chapter Outline. Section 6.1.9 describes the tool and the individual features of CodeInspect. In Section 6.2 we describe the usage of CodeInspect on an a real malware investigation. Section 6.3 concludes this chapter.

6.1 ARCHITECTURE

CodeInspect is a tool that is based on the Android frontend of the Soot framework. Different parts of the tool make use of algorithms implemented in Soot such as the use-def chain analysis or different callgraph algorithms. The GUI of CodeInspect is implemented with Eclipse' Rich Client Platform[1]. Eclipse provides different toolsets for developing Android applications including source code editors for code completion or renaming of variables, debugging support or Android device interaction with the Dalvik Debug Monitor Server (DDMS). These features are very useful during an app development process. However, most of them are also useful for manually reverse engineering apps. The key difference is the language with which the user or analyst is interacting. The source code in case of app development and the bytecode for reverse engineering tasks. CodeInspect's language is a variation of Soot's internal intermediate representation called Jimple [ARB13]. The framework includes different features that support an analyst with insights about the behavior of an app. In the following, we explain the main features and focus on the SuSi, HARVESTER and FuzzDroid (described in previous chapters) features.

6.1.1 *Main Components*

Figure 19 shows the start screen once CodeInspect successfully imported an apk either from the file system or directly from the smartphone. In this example, CodeInspect imported the Android/BadAccents[2] malware and shows an excerpt of the AndroidManifest. Under *Project Explorer* on the left side, an analyst can inspect all the files that are part of the apk in a human readable format. The Jimple files are located in the *Sources* folder. This figure furthermore shows that CodeInspect comes with different *Perspectives* and *Views*. A perspective defines different parts of the UI in which a view is a specific part of. For instance, SuSi, HARVESTER or FuzzDroid are implemented in form of a view, while the debugger or the dataflow analysis support is implemented in an own perspective. Depending on the goal of an analyst, she can switch between these perspectives.

1 https://wiki.eclipse.org/Rich_Client_Platform
2 MD5: a5028fd5df93ba753d919f02b7bf1106

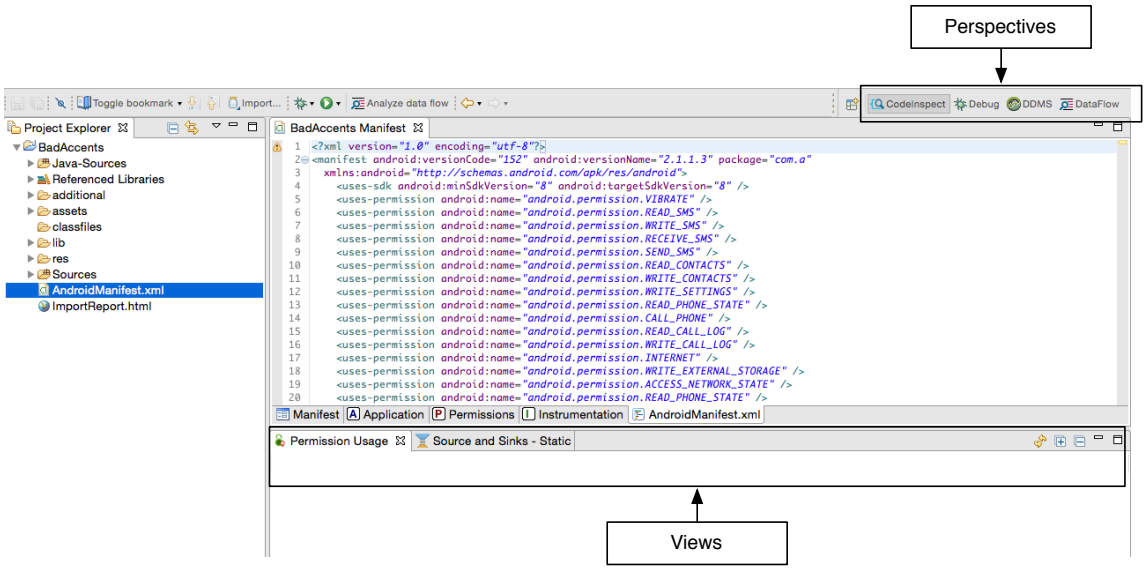

Figure 19: Overview of CODEINSPECT

6.1.2 *Standard Features*

The bytecode editor represents every class in an intermediate representation called Jimple. For better readability, we modified the original Jimple representation. For instance, we striped off the fully-qualified name for classes and instead only show the class name and import the package name in the beginning of the class. Every class file is represented with syntax highlighting and the analyst is able to rename identifiers such as method names or variable names.

A very useful feature during a malware investigation is the *Open Call Hierarchy* feature that lists the callgraph for a specific statement in the code. This is especially useful if one needs to determine initial triggers, e.g., incoming SMS messages, that may lead to the corresponding statement (influenced by conditions on the path). Figure 20 shows a callgraph from the `sendTextMessage()` API call to different initial triggers. It contains two `onReceive()` callbacks that are triggered once the application receives an incoming SMS (`com.a.a.AR`) or receives an incoming call (`com.shit.service.CallService$1`). The third callback `onClick()` gets triggered once the user clicks on a certain button. If one of these callbacks get triggered, it is very likely (depending on path conditions) that an SMS message gets sent.

Another handy feature is the field and variable *read* and *write* access tracking. This gives an analyst the possibility to find all statements that read a particular variable or field and on the other hand all statements that write to a variable or field.

During a malware investigation, it is important to document findings. This is especially useful for long investigations or investigations that involves different analysts. Comments on certain code statements make it easier to understand the important findings of an analyst. For that purpose, we use the *Bookmark* feature of Eclipse.

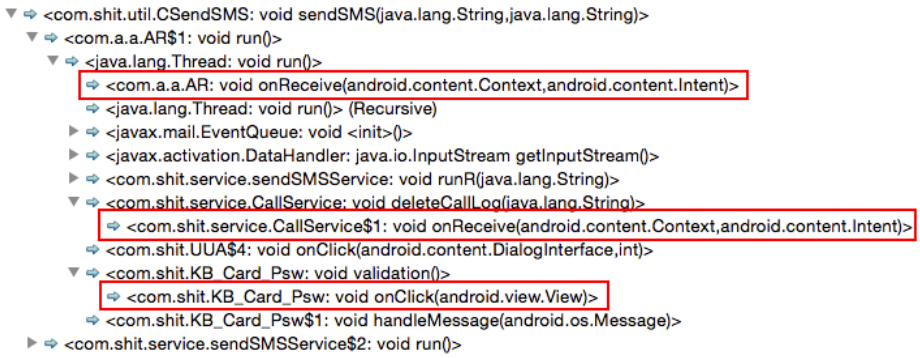

Figure 20: Call hierarchy for the sendTextMessage API call

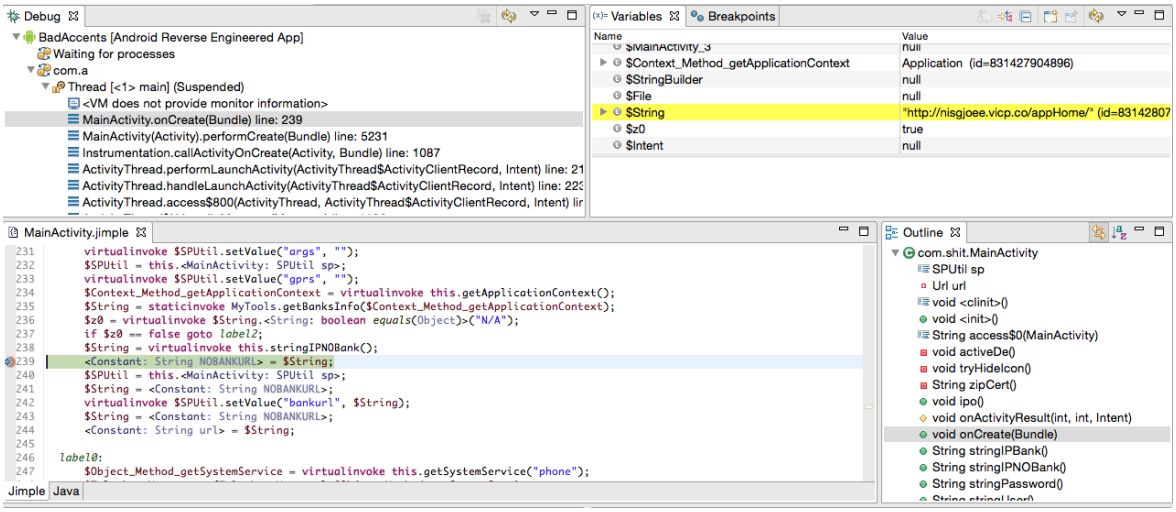

Figure 21: Overview of CodeInspect's Jimple debugger perspective

6.1.3 *Jimple Debugging*

Figure 21 shows CODEINSPECT's Jimple debugger perspective. It is very similar to a standard Eclipse debugger perspective for Java or Android applications. The only difference is the representation language of the code, which is in our case a modified version of Jimple. The analyst is able to set breakpoints at Jimple code locations and once hit, can step through the code in a single-step mode for instance. Figure 21 also shows the *Variables* view, which contains the runtime values for every variables. In the example, one can see that the variable-name `$String` points to the string `"http://nisgjoee.vicp.co/appHome/"`, a malicious website. Due to Jimple's type-based language, we are able to represent complex data structures such as an object with different fields. In comparison to other debugger such as IDAPro [Ida], which usually operates on an untyped intermediate representation, it is in general not possible to display runtime values of complex data structures if there are no type information of the object. Apart from the runtime value information, one can also modify the runtime values in the *Variables* view. Furthermore, Figure 21 also shows the stack frames (*Debug* view) at the current statement (line 239). This is very useful if one needs to track the sequence of nested functions.

6.1.4 FLOWDROID *Plugin*

In many malware investigations, it is useful to know *what* kind of data are leaked *where* or where particular pieces of data flow into. FlowDroid [Arz+14b] is a context-, flow-, field-, object-sensitive and lifecycle-aware dataflow-tracking tool for Android Apps. It is very precise and supports an analyst answering these questions. FlowDroid comes with a lot of individual settings [Arz16] that gives an analyst the option to fine-tune her analysis. Modifying these settings usually results in a faster analysis, which can negatively influence the precision and recall. We run FlowDroid on a *Class Hierarchy Analysis (CHA)* based callgraph that excludes exceptional paths. Furthermore, we also disabled support for *implicit flows* [Kin+08] and used a flow-insensitive alias analysis. These settings are not the most precise ones, but based on our experience, the ones which scale best for current Android malware applications. The specification of Android source and Android sink (see Section 3.3) methods can be regulated by the analyst. However, CODEINSPECT comes with a pre-defined list of source and sink methods based on the results from SUSI (see Chapter 3).

6.1.5 *Permission-Usage View*

Android applications usually contain different *Resource Methods* (see Section 3.3) such as accessing location information or sending SMS messages. Most of these resource methods are protected by permissions and need to be declared in the AndroidManifest by the app developer. For instance, if the developer wants to use the `sendTextMessage()` API method, she needs to declare an `android.permission.SEND_SMS` permission in the AndroidManifest. Unfortunately, due to a lack of documentation of the AOSP, there is little information about the mapping between the resource methods and the corresponding permissions. Therefore, researchers proposed different approaches [Au+12; Fel+11] that try to extract that mapping. Based on the results of the latest research [Au+12], we also offer a permission-method view in CODEINSPECT. Figure 22 shows an example of the view. Please keep in mind that the example shows original statements of the Android/BadAccents malware. One can see

that the malware needs to implement the `android.permission.GET_TASKS` permission in order to use the sensitive `getRunningTasks()` method. This API returns a list of tasks that are currently running, which is usually used for tapjacking attacks [Ras+15b] in malicious applications.

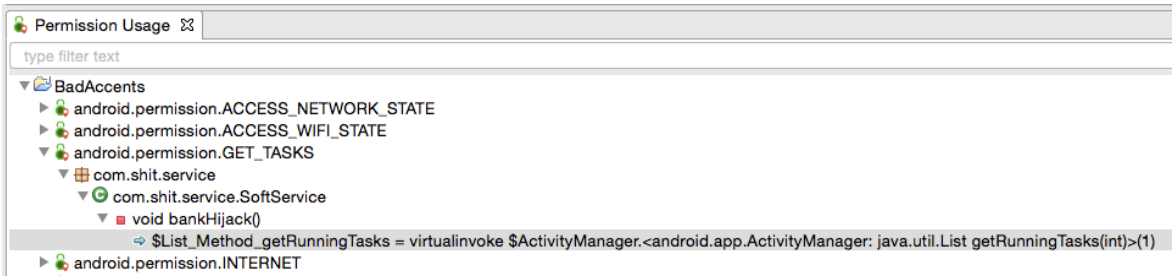

Figure 22: Overview of CodeInspect's Permission-Usage view

6.1.6 *Communications View*

In many malware investigations, it is essential to know the different communication channels and more important the concrete addresses of these channels. Examples are concrete URLs or SMS numbers. Therefore, CodeInspect contains a view called *Communications*, which provides concrete information about the addresses and the corresponding API calls where these addresses are used.

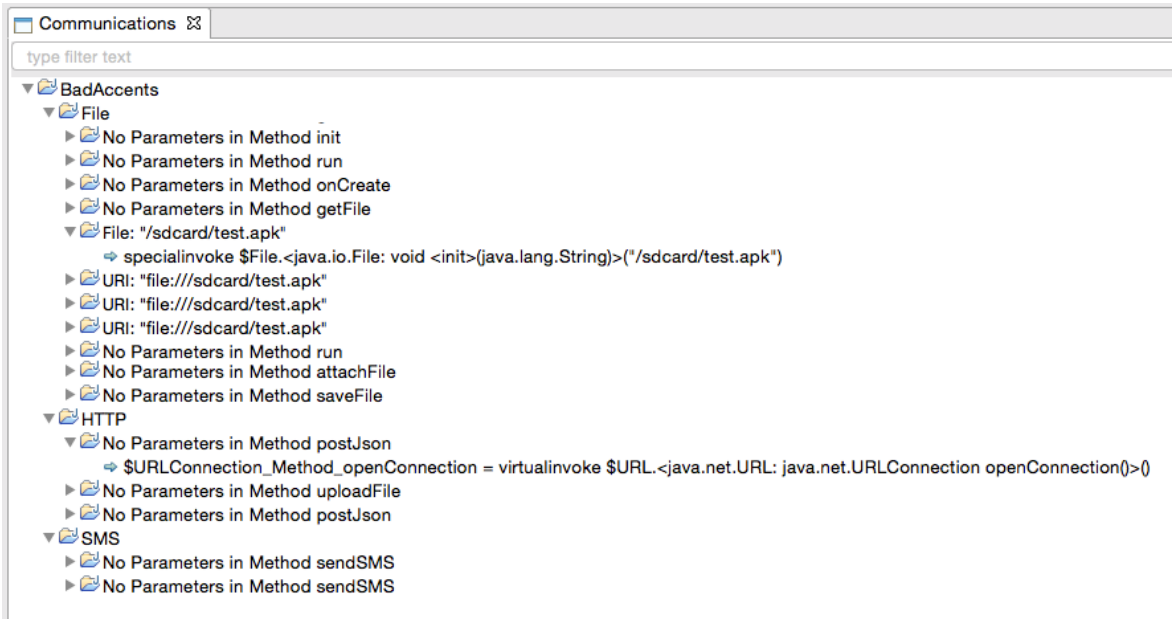

Figure 23: CodeInspect's Communications view

This information is statically extracted with the help of FlowDroid's inter-procedural constant string propagation approach. Different Android sink (see Section 3.3) API calls are not directly called with a constant string as address. For instance, the `HttpClient.execute(param1)` API can be called with an `HttpGet` object (`param1`), where the constant URL string gets passed into the `HttpGet` constructor. Since we want to provide the analyst the actual Android sink API with the corresponding string address, we applied an

additional static dataflow tracking analysis to get the connection between the API that gets passed the constant string and the actual Android sink API.

Figure 23 shows an example of this view. It categorizes the findings into different categories, such as *File*, *HTTP* or *SMS* in our malware example. Figure 23 furthermore shows that a static extraction of information about different communication channels was only possible in a small number of cases.

If the information is not statically extractable, mainly due to static-analysis limitations, we apply the HARVESTER approach (see Section 6.1.8).

6.1.7 SuSi *View*

As described in Chapter 3, we proposed a new approach for automatically identifying Android source and Android sink API methods. Furthermore, we also categorize these source- and sink-API calls.

Information about Android sources and Android sinks are not only important for a dataflow analysis approach, they are also important for a malware investigation. Android sources give a good overview what kind of sensitive data are accessed via different resource methods and Android sinks give insights about different communication channels, like Internet or email. CodeInspect offers this information to the analyst in form of a view as shown in Figure 24.

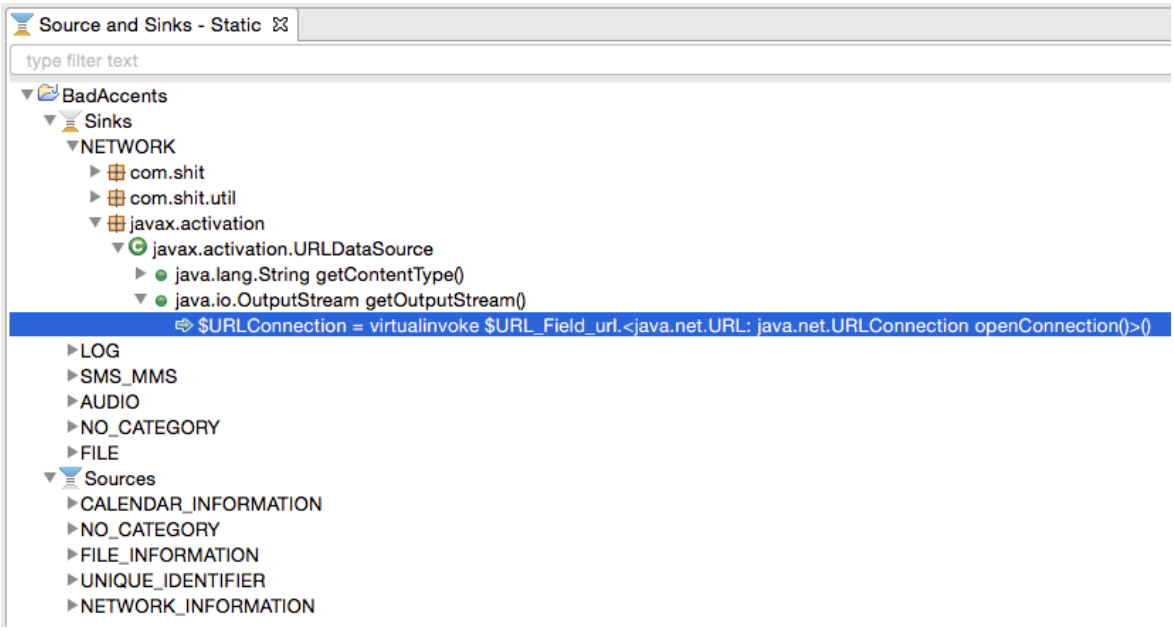

Figure 24: Overview of CodeInspect's Sources and Sinks view

6.1.8 HARVESTER *Integration*

In Chapter 4 we introduced HARVESTER, a hybrid code analysis approach that extracts runtime values at concrete code locations in the application's bytecode. HARVESTER is able to extract these values even in highly obfuscated applications that try to circumvent static and dynamic code analysis approaches. Depending on the application, extracting

these runtime values may take some time. Since manual investigations are usually limited in time, we only apply HARVESTER in cases where a purely static or dynamic analysis is not sufficient enough. For instance, the *Communications* view applies static code analysis approaches for extracting concrete addresses. Only if the analysis is not able to extract these values, we apply HARVESTER as a post-analysis.

Apart from the extraction of runtime values, Section 4.3.2.3 describes another scenario where HARVESTER can be applied: the resolution of reflective method calls. This feature is also integrated into CodeInspect. The analyst simply needs to click on the reflective method call that needs to be resolved and HARVESTER gets applied. In case of a successful resolution, HARVESTER re-writes the Jimple code and inserts information about the original API call (see Section 4.3.2.3).

6.1.9 FuzzDroid *View*

The decision whether an application is malicious or not is in many cases not easy to answer. For instance, if an application displays an alert dialog to the user, this can be outright benign behavior, but it can also be used for malicious purposes. Examples are malware samples that checks whether a specific banking application is installed on the device and only then displays a UI that prompts the user to add her banking credentials [Ras+15b]. Information about the environment (see Section 5), for instance, under what circumstances the UI gets displayed, helps the analyst to judge about the maliciousness.

For that purpose, we will integrate FuzzDroid into CodeInspect in form of a new view in the near future. The analyst will be able to click on an interesting code location and FuzzDroid tries to identify all environments that are necessary to reach that particular code location. A graphical example of this view will be described in Section 6.2.

6.2 APPLICATION SCENARIO: INVESTIGATION OF THE ANDROID/BADACCENTS MALWARE

In the main motivating example (see Section 1.1.1), we introduced our malware investigation together with Intel Security [Ras+15b; Ras+15a]. We discovered a new malware family (Android/BadAccents) that had already infected more than 20.000 devices before we stopped the threat. Back then, we had to manually reverse engineer the malicious sample in order to identify the different communication channels of the malware and to understand the concrete behavior of the malware. The complete investigation was very time consuming since little approaches existed that could support us during the investigation.

However, in this dissertation, we proposed three different approaches SuSi (Chapter 3, HARVESTER (Chapter 4) and FuzzDroid (Chapter 5) that provide concrete insights about the behavior of an application. All of them are integrated into CodeInspect that makes it convenient for an analyst to inspect a malicious application. In the following, we demonstrate how CodeInspect can be used during a real malware investigation and what kinds of insights of the application are presented to the analyst. We will exemplarily explain this on the Android/BadAccents malware.

Insights Through SuSi *and Jimple Debugger*

The *Source and Sink* view provides a first overview of Android sources and Android sinks that are used in the app. This can be an initial starting point for a deeper investigation. For instance, Figure 25 shows that the application makes use of the `getInstalledApplications()` API call. This API call returns a list of all applications that are installed on the device and does not require any permission in the AndroidManifest. Therefore, it would not appear in the *Permission Usage* view, but SuSi's list contains this sensitive Android source API call.

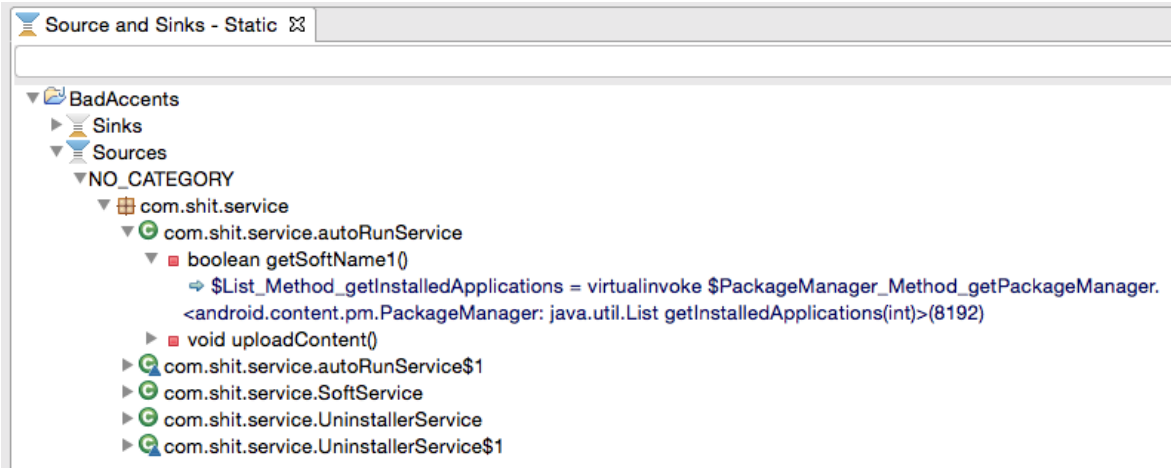

Figure 25: Sources and Sinks view shows access to information about installed applications

A double-click on the statement in the *Source and Sink* view directly jumps to the code location in the Jimple editor. From there, we can now start with a deeper investigation why the malware is accessing this information. Within CodeInspect there are different options to do so. One example would be the debugging mode where we set a breakpoint at that code location and once hit, we can dynamically debug the application in a single-step mode. This gives us the opportunity to follow the execution path to understand why the malware needs to access this sensitive data.

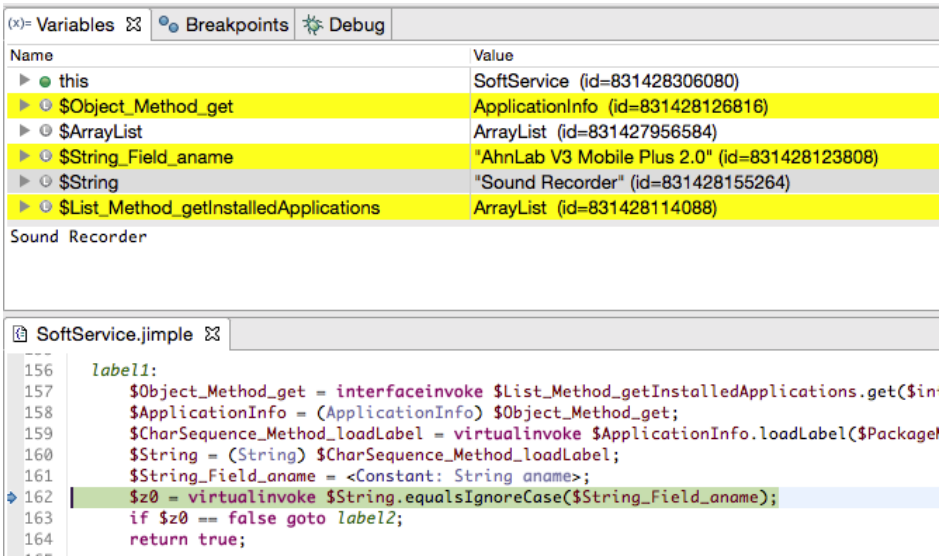

Figure 26: Usage of Jimple debugger on the Android/BadAccents malware

CodeInspect executes the application and switches into the debugging mode. Fortunately, the breakpoint gets immediately hit, once the application is opened. From there on, we continue the debugging process until we reach the first position that reads the data from the `getInstalledApplications()` call. Figure 26 shows the code location (`equalsIgnoreCase()`) and with the help of the *Variables* view inside the debugger perspective, we see that the malware checks all installed applications (currently the *Sound Recorder* application) if it is equals to the "AhnLab V3 Mobile Plus 2.0" application. This is a security application from a company located in South Korea. A further debugging process reveals that the application tries to uninstall the security application if it is installed on the device. The malware is probably afraid that it gets detected by the security application, in case it is installed on the device. These are very interesting insights about the behavior of the application provided by SuSi's output together with the Jimple debugger.

Insights Through HARVESTER

Since the malware sample contains some packages of the type `javax.mail.*`, we assumed that the malware uses email-communication for stealing sensitive data, which would be back then unique for Android malware. Using the text search feature of CodeInspect with "mail" revealed that the malware implements a method with the name `MailSend()`. The body of that method revealed that two of the arguments are the username and the password of the mail account, which are necessary for sending emails using that account. Concrete values for both credentials are not easy to extract since they are encoded within native code [Ras+15b]. Therefore, one can use HARVESTER with the two arguments as logging points and it automatically extracts the runtime values, even if the code is obfuscated. HARVESTER successfully extracted the username `hjgyfjhg1010@126.com` and password `zxcv1234` without any human interaction. These two credentials were sufficient for a further email-account investigation, which revealed the impact of the malware, 20.000 infections within two month. It further helped for a legal investigation of the malware authors. This shows, that the automatically extraction of runtime values are very important during a malware investigation.

Insights Through FuzzDroid

CodeInspect's *Permission Usage* view reveals that the malware sample implements some functionality for sending SMS messages (see Figure 27). This can be used for benign reasons, but also for malicious reasons. In many cases, it is essential to know the environment in which the malware has to run for determining whether the sending of the SMS message is malicious or not.

The FuzzDroid view provides more insights about the environment. Figure 28 shows that SMS messages are only sent once the device first receives an SMS with a concrete message body of the form `sd_aaaaaaaB`. Furthermore, there has to be also one contact stored on the device. If both conditions are met, an SMS with the content `aaaaaaaB` is sent to that contact. The `aaaaaaaB` part of the incoming SMS messages is a string that was automatically generated by the Symbolic Value Provider (see Section 5.3.1) and is usually not sent by an attacker. Since FuzzDroid is a fuzzing based approach, it provides one possible value that could have been sent. In a more generic way, we can say that the incoming SMS message has to have the form `sd_<Attacker Text>` (e.g., `sd_aaaaaaaB`). This

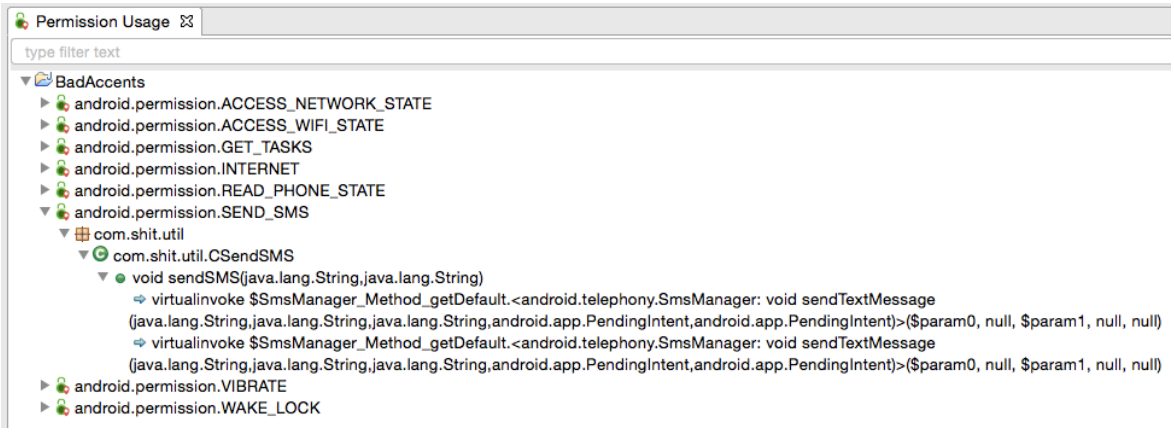

Figure 27: Permission view shows the usage of sendTextMessage

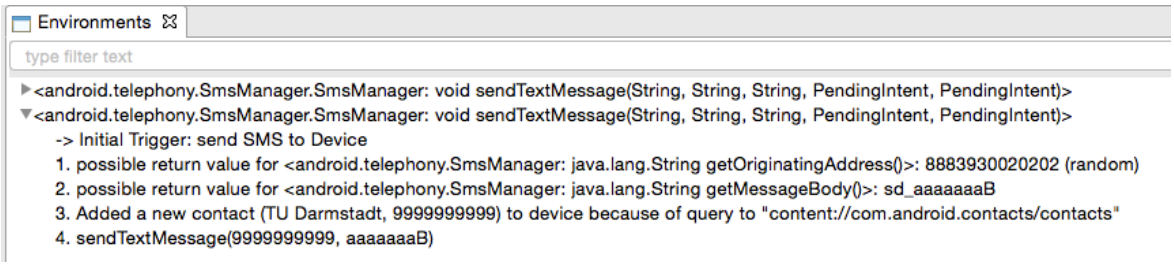

Figure 28: Preview of Environment view in CodeInspect

is a very common pattern for C&C communication. The format is usually of the form `ACTION SEPARATOR SETTING`, which is in our case `sd _ aaaaaaaB`. Figure 28 further shows that the `aaaaaaaB` string is part of the SMS body in the 4th step. Step 3 reveals that the malware accesses a contact (TU Darmstadt with the phone number 9999999999), where the number of that contact is also used for sending an SMS message in step 4. All these steps are a very common pattern of SMS phishing attacks (see Section 2.2) that sends an SMS message to all contacts stored on the device. Based on the provided information, it is very likely that the attacker controls the text of the SMS spam, since the `aaaaaaaB` message is received from an incoming SMS message (likely to be a C&C communication), which is also used for sending the SMS message in step 4.

In summary, this shows that the FuzzDroid view provides a lot of important facts that provide detailed information to the analyst who needs to decide whether an application contains malicious behavior or not. In this particular case, we manually verified it and can say that FuzzDroid was indeed able to point an analyst to an SMS phishing attack. Even more, FuzzDroid was able to identify the concrete C&C communication protocol (`sd_<Phishing Text>`) for sending SMS spam. With the help of FuzzDroid, an analyst does not have to wait until the C&C server sends a command to the victim, she can immediately extract environment information without any human or server interaction. This saves a lot of time, and helps speeding up the manual investigation.

6.3 SUMMARY AND CONCLUSION

In this chapter, we introduced the implementation of our novel reverse engineering framework called CodeInspect. It converts the Android bytecode into a human readable interme-

diate representation called Jimple and contains different features, which provide detailed information about the behavior of an application. This information is essential during an investigation.

More concretely, we demonstrated how SuSi, HARVESTER and FuzzDroid reveal detailed insights about the behavior of an application during a real malware investigation. We explained this using the Android/BadAccents malware family and showed different insights of the malware, which were not known prior to this work, only after applying these new approaches. Insights about this malware sample show the complexity of modern Android malware. Without the help of approaches that combine static with dynamic code analysis approaches, it would have been much harder to extract these insights.

7 DISCUSSION AND CONCLUSION

In this dissertation, we addressed the problem of automatically extracting fundamental insights about the behavior of an Android application. More concrete, we focused on the automatic identification of sensitive API calls from the Android framework (Chapter 3), the automatic extraction of runtime values from the application's bytecode (Chapter 4), the automatic de-obfuscation of reflective method calls (Chapter 4) and the automatic extraction of context information (environment) under which a certain code location gets reached (Chapter 5). These insights are very important during a malware investigation, which would have, prior to this work, required a time-consuming, manual effort. Also existing, automated approaches from literature would have not been able to extract these insights in highly obfuscated malware applications that apply anti-static and anti-dynamic code obfuscation techniques.

7.1 SUMMARY OF CONTRIBUTIONS

The CodeInspect framework described in this dissertation consists of different approaches, SuSi, HARVESTER and FuzzDroid, for automatically identifying concrete insights about the behavior of an application, even if the application's code is highly obfuscated.

In Chapter 3, we presented an approach called SuSi for automatically identifying sensitive source and sink API methods in the Android operating system. The approach is based on machine-learning, which provides a categorized list of sensitive source and sink API methods of a specific Android version. Therefore, the whole Android API gets transformed into a smaller set of relevant sensitive API methods that read (source) or write (sink) from/to Android resources. SuSi's output is used to improve the lack of completeness of source and sink lists used by different automated code analysis approaches [Avd+15; Arz+14b; Liu+15] that try to extract insights of the behavior of an application. It also serves as a starting point in manual malware investigations helping an analyst to identify malicious behaviors. SuSi is an important element in the CodeInspect framework, since it directly supports HARVESTER (Chapter 4) and FuzzDroid (Chapter 5) with sensitive API methods that are essential for identifying concrete runtime values or the environment under which an application reaches a certain code location, respectively.

Chapter 4 addresses the problem of automatically extracting runtime values at any code location (concrete values of API-arguments) in an Android application. The automatic extraction of runtime values is especially important at code locations that provide important insights of an application, like the connection to a server, the sending of SMS messages or the sending of emails. The approach is called HARVESTER and combines static and dynamic code analysis techniques in such a way that it is able to automatically extract runtime values even in applications that use obfuscation techniques against static and/or dynamic code analysis approaches. Our evaluation showed that HARVESTER is not only able to extract runtime values with a perfect precision, it is able to extract these values within minutes. This stresses the practical feasibility of this approach. The HARVESTER approach is further able to automatically resolve reflective method calls in cases where re-

flective method calls are used as an obfuscation technique. We further evaluated how HARVESTER can be used to create precise intra- and inter-component callgraphs. Especially the latter one is one of the biggest limitation in current whole static analysis approaches for Android application. However, we have shown that HARVESTER is able to reduce the amount of callgraph edges significantly, outperforming current state-of-the-art approaches. The techniques applied by HARVESTER can also be used to support dynamic taint tracking approaches in immediately identifying a data leak in cases where anti-dynamic code obfuscation techniques are applied. As one can see, HARVESTER is a fundamental element in the CodeInspect framework, which can be often used as a standalone tool supporting other static or dynamic code analysis approaches, such as FuzzDroid (Chapter 5).

In Chapter 5, we introduced an approach that automatically extracts the context, i.e., the environment, under which a certain code location gets reached. FuzzDroid, a fuzzing-based approach, gives insights about the conditions, e.g., receiving a specific SMS message, an application expects to execute a certain behavior. The output of this approach provides concrete context-insights that further help to identify malicious activities in applications. The approach consists of different static and dynamic code analysis approaches, which makes it possible to analyze applications containing anti-static and anti-dynamic code obfuscation techniques. Our evaluation resulted that pure static and pure dynamic code analysis approaches are not sufficient enough for extracting environment-information from modern Android malware applications.

It is important to highlight that the SuSi, HARVESTER and FuzzDroid approach are highly connected to each other. Without the pre-analysis of SuSi, HARVESTER and FuzzDroid would not be supported with sensitive API methods. On the other hand, HARVESTER's output, the automatic de-obfuscation of reflective method calls is essential for FuzzDroid, which otherwise would not be able to determine a target location. In summary, only a combination of SuSi, HARVESTER and FuzzDroid performs the best insights about the behavior of modern malware applications that include anti-static and anti-dynamic code obfuscation techniques.

7.2 CONCLUSIONS

This research confirmed that it is possible to automatically extract fundamental insights about the behavior of highly obfuscated Android applications. In particular:

- It has shown that a machine-learning based approach (Chapter 3) is able to automatically identify sensitive API methods that read (sources) and write (sinks) from/to resources in the Android framework. Furthermore, the same approach is also able to automatically categorize these source and sink API methods. [Thesis Statement TS-1].

- It has described that a combination of static program slicing with code generation and concrete dynamic code execution (Chapter 4) is required to extract runtime values from highly obfuscated (Section 1.1.2) Android applications [Thesis Statement TS-2]. The same technique can be also applied to create precise inter-component (Android inter-component communication) callgraphs (Chapter 4.4.2) [Thesis Statement TS-4].

- It has shown that a combination of static program slicing with code generation, concrete dynamic code execution and bytecode manipulation (Section 4.3.2.3) is able to

resolve reflective method calls within highly obfuscated Android application that is implemented as a form of static code obfuscation [Thesis Statement TS-3].

- It has shown that a combination of static program slicing with code generation (Chapter 4) can be also used as a pre-analysis for dynamic taint tracking approaches in order to immediately identify a data leak [Thesis Statement TS-5].
- It has described that a fuzzing based approach (Chapter 5) that makes use of different static and dynamic code analysis techniques, in combination with an evolutionary algorithm, is able to identify conditions under which a certain code location is reached, especially in those cases were an application applies anti-analysis techniques described in Section 1.1.2 [Thesis Statement TS-6].

All the above confirms that extracting fundamental insights about the behavior of a potential malicious application that applies different code obfuscation techniques (Section 1.1.2) is only possible if an analysis combines static with dynamic code analysis approaches.

7.3 PRACTICAL IMPACT OF THE CODEINSPECT FRAMEWORK

The CODEINSPECT framework described in Chapter 6, has already had a large impact on the security setup of different companies and app developers. As already mentioned in Section 1.6, CODEINSPECT helped identifying two serious Android operating system bugs [Arz+14a; Ras+15b] and helped identifying a new Android malware family [Ras+15b; Ras+15a]. Furthermore, it also helped in discovering a major data leakage in Backend-as-a-Service solutions [RA15] of Facebook and Amazon.

This shows that the theoretical concepts in the individual parts of the CODEINSPECT framework work very well in practice. Furthermore, it already helped in identifying serious security problems in individual Android applications, which were not known prior to this work.

7.4 FUTURE RESEARCH DIRECTIONS

Automatic Malware Detection The focus of this dissertation was on automatically extracting insights of the behavior of an application in the context of malware-investigations. However, we did not focus on automatically judging whether an application is malicious or not. Based on the results of SUSI, HARVESTER and FUZZDROID, the next step in this research area would be to combine these results to automatically identify malicious activities in applications. There are already first research approaches in this area that are based on machine-learning [Gor+14; Cha+13; Arp+14] but their feature set relies on features that expect more or less un-obfuscated applications. Combining these approaches with the results of this dissertation is an interesting part for future research.

Application to Native Code The implementation of SUSI, HARVESTER and FUZZDROID currently focuses on Java-based Android code. However, Android applications or parts of an application can also be implemented in C/C++ [Deva]. Malware applications can also implement its malicious behavior within the native part of an application, causing one to extract less precise behavior information with our proposed approaches. Fortunately, the

majority of current Android malware is implemented in Java and only a few malicious samples make use of native code. However, we see a trend, especially for commercial packers that adopted their unpacking process from a previous Java-based implementation [LY16] to a native-based implementation [Dex]. Moreover, there are also a few malware samples [Li13] that implement its malicious behavior in the native layer. Therefore, it is an interesting research question if the proposed approaches can be also applied to native code in Android applications and how efficient such approaches would be.

8

ABOUT THE AUTHOR

Siegfried Rasthofer is a research assistant at the TU Darmstadt and Fraunhofer SIT, Germany. He received his Master of Science (IT-Security) in 2012 at the University of Passau and his Bachelor of Science in 2010 at the University of Applied Sciences Landshut. His main research focus is on applied software security on Android applications. He developed different tools that combine static and dynamic code analysis for security purposes. Most of his research is published at academic as well as industry conferences.

Awards

- 1st place of German Prize for IT-Security 2016 (together with Steven Arzt, Marc Miltenberger and Eric Bodden)
- Technology Review Best Innovator under 35 (together with Steven Arzt)
- Second prize of HIGHEST Startup Contest (together with Steven Arzt, Marc Miltenberger and Eric Bodden)
- Google Patch Reward 2015 for Reporting and Fixing a Security Vulnerability in the AOSP (together with Stephan Huber)

Involved Projects

- Finding and Demonstrating Undesired Program Behavior (TESTIFY), DFG
- Associated Project at the Priority Program Reliably Secure Software Systems - RS3 (Phase 2)
- RUNSECURE (DFG Emmy Noether Program)
- ZertApps - Certified Security of Mobile Apps (BMBF project)

Supervised Theses

- A Machine-Learning Approach to Automatically Identify App Categories Based on Android Binaries (Master thesis by Marc Miltenberger)
- Detecting Android Obfuscation Techniques (Master thesis by Julien Hachenberger)
- Android Malware Detection by Machine Learning (Bachelor thesis by Patrick Pauli)
- Statische Informationsflussanalyse mittels symbolischer Ausführung für Android (Bachelor thesis by Robert Hahn)
- Slicing-basierte String-Extraktion in Androidapplikationen (Bachelor thesis by Marc Miltenberger)

- Hybrid Inter-Component and Inter-Application Data Flow Analysis in Android (Master thesis by Dieter Hofmann)
- Dynamically Enforcing Usability and Security Properties of Android Advertisement Libraries (Bachelor thesis by Max Kolhagen)

Peer-Reviewed Publications

[ARB13] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. "Instrumenting Android and Java Applications as Easy as abc." In: *Runtime Verification (RV)*. Lecture Notes in Computer Science. Springer, 2013.

[Arz+13] Steven Arzt, Kevin Falzon, Andreas Follner, Siegfried Rasthofer, Eric Bodden, and Volker Stolz. "How Useful Are Existing Monitoring Languages for Securing Android Apps?" In: *Software Engineering (Workshops)*. 2013.

[Arz+14a] Steven Arzt, Stephan Huber, Siegfried Rasthofer, and Eric Bodden. "Denial-of-App Attack: Inhibiting the Installation of Android Apps on Stock Phones." In: *Proceedings of the Fourth ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. Nov. 2014.

[Arz+14b] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps." In: *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM. June 2014.

[Arz+15] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. "Using Targeted Symbolic Execution for Reducing False-positives in Dataflow Analysis." In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2015. ACM, 2015.

[Avd+15] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. "Mining Apps for Abnormal Usage of Sensitive Data." In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE 2015. ACM, 2015.

[Eli+16] Nicole Eling, Siegfried Rasthofer, Max Kolhagen, Eric Bodden, and Peter Buxmann. "Investigating Users Reaction to Fine-Grained Data Requests: A Market Experiment." In: *49th Hawaii International Conference on System Sciences (HICSS)*. IEEE Computer Society, 2016.

[HR16] Stephan Huber and Siegfried Rasthofer. "How to do it Wrong: Smartphone Antivirus and Security Applications Under Fire." In: *Def Con 24*. Aug. 2016.

[HRA16] Stephan Huber, Siegfried Rasthofer, and Steven Arzt. "(In-) Security of Smartphone AntiVirus and Security Apps." In: *VirusBulletin 2016*. Oct. 2016.

[Li+15] Li Li, Alexandre Bartel, Tegawende Bissyande, Jacques Yves Klein, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps." In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE 2015. ACM, 2015.

[RA15] Siegfried Rasthofer and Steven Arzt. "(In-)Security of Backend-As-A-Service Solutions." In: *Blackhat Europe*. Nov. 2015.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks." In: *2014 Network and Distributed System Security Symposium (NDSS)*. 2014.

[RCH15] Siegfried Rasthofer, Carlos Castillo, and Alex Hichliffe. "We know what you did this Summer: Android Banking Trojan Exposing its Sins in the Cloud." In: *18th Association of Anti-virus Asia Researchers International Conference (AVAR) 2015*. Dec. 2015.

[Ras+14] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. "DroidForce: Enforcing Complex, Data-Centric, System-Wide Policies in Android." In: *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*. IEEE. Sept. 2014.

[Ras+15a] Siegfried Rasthofer, Steven Arzt, Max Kolhagen, Brian Pfretzschner, Stephan Huber, Eric Bodden, and Philipp Richter. "DroidSearch: A Powerful Search Engine for Android Applications." In: *2015 Science and Information Conference (SAI)*. 2015.

[Ras+15b] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. "How Current Android Malware Seeks to Evade Automated Code Analysis." In: *9th International Conference on Information Security Theory and Practice (WISTP'2015)*. 2015.

[Ras+16a] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques." In: *2016 Network and Distributed System Security Symposium (NDSS)*. San Diego, 2016.

[Ras+16b] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. "Reverse Engineering Android Apps With CodeInspect (invited paper)." In: *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security (IMPS)*. 2016.

[Ras+17] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. "Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments." In: *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. ACM, 2017.

[Ric+13] P. Richter, E. Bodden, S. Rasthofer, and A. Roßnagel. "Schutzmaßnahmen gegen datenschutzunfreundliche Smartphone-Apps - Technische Möglichkeiten und rechtliche Zulässigkeit des Selbstdatenschutzes bei Apps, DuD 2013." In: *DuD* (2013).

Technical Reports

[ARB13] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. *SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks*. Tech. rep. TU Darmstadt, 2013.

[Avd+14] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. *Mining Apps for Abnormal Usage of Sensitive Data*. Tech. rep. Saarland University and TU Darmstadt, 2014.

[Fri+13] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. *Highly Precise Taint Analysis for Android Application*. Tech. rep. TU Darmstadt, 2013.

[Li+14] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. *I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis*. Tech. rep. University of Luxembourg, TU Darmstadt, and Pennsylvania State University, 2014.

[Li+16] Li Li, Tegawendé François D Assise Bissyande, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. *Static Analysis of Android Apps: A Systematic Literature Review*. Tech. rep. University of Luxembourg, Fraunhofer SIT/TU Darmstadt, University of Wisconsin and Pennsylvania State University, Apr. 2016.

[Ras+15a] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. *An Investigation of the Android/BadAccents Malware which Exploits a new Android Tapjacking Attack*. Tech. rep. TU Darmstadt and McAfee Research Lab, Apr. 2015.

[Ras+15b] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. *Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques*. Tech. rep. TU Darmstadt, 2015.

Posters

[Arz+15] Steven Arzt, Alexandre Bartel, Richard Gay, Steffen Lortz, Enrico Lovat, Heiko Mantel, Martin Mohr, Benedikt Nordhoff, Matthias Perner, Siegfried Rasthofer, et al. "Poster: Software security for mobile devices." In: *Poster at the 36th IEEE Symp. on Security and Privacy*. 2015.

[Arz+16] Steven Arzt, Alexandre Bartel, Richard Gay, Steffen Lortz, Enrico Lovat, Heiko Mantel, Martin Mohr, Benedikt Nordhoff, Matthias Perner, Siegfried Rasthofer, et al. "Poster: Software security for mobile devices." In: *Poster at USENIX Security 2016*. 2016.

BIBLIOGRAPHY

[Jdg] *A standalone Java Decompiler GUI*. https://github.com/java-decompiler/jd-gui. Github project. 2016.

[Enc] *A study of android application security - Fortify Rules*. http://www.enck.org/tools/fsca_rules-final.xml. Apr. 2013.

[Apk] *A tool for reverse engineering Android apk files*. https://ibotpeaches.github.io/Apktool/. 2016.

[Abr+15] A. Abraham, Radoniaina Andriatsimandefitra, A. Brunelat, Jean-François Lalande, and V. Viet Triem Tong. "GroddDroid: a gorilla for triggering malicious behaviors." In: *Proceedings of the 2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. MALWARE '15. IEEE, 2015.

[Appa] *Activities*. https://developer.android.com/reference/android/app/Activity.html. last checked: Oct. 2016.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. "Detecting Equality of Variables in Programs." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. ACM, 1988.

[Ana+12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. "Automated Concolic Testing of Smartphone Apps." In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. ACM, 2012.

[Chaa] *Android API Differences Report*. https://developer.android.com/sdk/api_diff/9/changes.html. 2013.

[Chab] *Android API Differences Report*. https://developer.android.com/sdk/api_diff/17/changes.html. 2013.

[Chac] *Android API Differences Report*. https://developer.android.com/sdk/api_diff/18/changes.html. 2013.

[Anda] *Android Security 2015 Year in Review*. https://static.googleusercontent.com/media/source.android.com/de//security/reports/Google_Android_Security_2015_Report_Final.pdf. Whitepaper. 2016.

[Andb] *AndroidHooker*. https://github.com/AndroidHooker/hooker. Github project. 2016.

[Andc] *Application Fundamentals*. https://developer.android.com/guide/components/fundamentals.html. last checked: Oct. 2016.

[Appb] *Application Security*. https://source.android.com/security/overview/app-security.html. last checked: Oct. 2016.

[AA14] Axelle Apvrille and Ange Albertini. "Hide Android Applications in Images." In: *Blackhat EU*. Sept. 2014.

[AN14] Axelle Apvrille and Ruchna Nigam. "Obfuscation in android malware, and how to fight back." In: *Virus Bulletin* (2014).

[AA15] L. Apvrille and A. Apvrille. "Identifying Unknown Android Malware with Feature Extractions and Classification Techniques." In: *The 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-15)*. IEEE, Aug. 2015.

[Arp+14] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket." In: *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[Arz16] Steven Arzt. "Static Data Flow Analysis for Android Applications." PhD thesis. TU Darmstadt, 2016.

[ARB13] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. "Instrumenting Android and Java Applications as Easy as abc." In: *Runtime Verification (RV)*. Lecture Notes in Computer Science. Springer, 2013.

[Arz+14a] Steven Arzt, Stephan Huber, Siegfried Rasthofer, and Eric Bodden. "Denial-of-App Attack: Inhibiting the Installation of Android Apps on Stock Phones." In: *Proceedings of the Fourth ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. Nov. 2014.

[Arz+14b] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps." In: *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM. June 2014.

[Arz+15] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. "Using Targeted Symbolic Execution for Reducing False-positives in Dataflow Analysis." In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2015. ACM, 2015.

[Au+12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. "PScout: analyzing the Android permission specification." In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS '12. ACM, 2012.

[Avd+15] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. "Mining Apps for Abnormal Usage of Sensitive Data." In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE 2015. ACM, 2015.

[Bac+13] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. "AppGuard: enforcing user requirements on android apps." In: *Proceedings of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'13. Springer-Verlag, 2013.

[Bac+14] Michael Backes, Sven Bugiel, Erik Derr, and Christian Hammer. *Taking Android App Vetting to the Next Level with Path-sensitive Value Analysis*. Tech. rep. Center for IT-Security, Privacy and Accountability (CISPA), Saarbrücken, 2014.

[Bac+16a] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. "On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis." In: *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Aug. 2016.

[Bac+16b] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. "R-Droid: Leveraging Android App Analysis with Static Slice Optimization." In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIACCS '16. ACM, 2016.

[Bag+16] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. "Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android." In: *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2016.

[Bar+15] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. "Static analysis of implicit control flow: Resolving Java reflection and Android intents." In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE '15. Nov. 2015.

[Bar+12] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot." In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*. SOAP '12. ACM, 2012.

[BH07] Michael Batchelder and Laurie Hendren. "Obfuscating Java: The Most Pain for the Least Gain." In: *Proceedings of the 16th International Conference on Compiler Construction*. CC'07. Springer-Verlag, 2007.

[Bat+11] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications." In: *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*. MALWARE '11. IEEE Computer Society, 2011.

[Bho+14] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. "Brahmastra: Driving Apps to Test the Security of Third-party Components." In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. USENIX Association, 2014.

[Bod+11] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. "Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders." In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ICSE '11. ACM, 2011.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. "RWset: Attacking Path Explosion in Constraint-Based Test Generation." In: *Proceedings of the 14th International Conference (TACAS)*. 2008.

[BS09] Martin Bravenboer and Yannis Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses." In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. 2009.

[Appc] *BroadcastReceiver*. `https://developer.android.com/reference/android/content/BroadcastReceiver.html`. last checked: Oct. 2016.

[Bru+15] Lucas Brutschy, Pietro Ferrara, Omer Tripp, and Marco Pistoia. "ShamDroid: gracefully degrading functionality in the presence of limited resource access." In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM. 2015.

[BZNT11] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: Behavior-based Malware Detection System for Android." In: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '11. ACM, 2011.

[CDE08a] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. USENIX, 2008.

[CDE08b] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. USENIX Association, 2008.

[Cad+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death." In: *ACM Trans. Inf. Syst. Secur.* (Dec. 2008).

[CR16] Haipeng Cai and Barbara Ryder. *Understanding Application Behaviours for Android Security: A Systematic Characterization*. Tech. rep. Department of Computer Science, Virginia Polytechnic Institute & State University, 2016.

[Cao+15] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework." In: *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*. 2015.

[Cha+12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code." In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. IEEE Computer Society, 2012.

[Cha+13] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. "MAST: Triage for Market-scale Mobile Malware Analysis." In: *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '13. ACM, 2013.

[Chi12] Eric Chien. *Motivations of Recent Android Malware*. `http://investor.symantec.com/files/doc_news/2012/motivations_of_recent_android_malware.pdf`. 2012.

[Chi+11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. "Analyzing inter-application communication in Android." In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. MobiSys '11. ACM, 2011.

[CNS13] Wontae Choi, George Necula, and Koushik Sen. "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning." In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. ACM, 2013.

[CGO15] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. "Automated Test Input Generation for Android: Are We There Yet?" In: *CoRR* abs/1503.07217 (2015).

[Chy15] Filip Chytry. *Apps on Google Play Pose As Games and Infect Millions of Users with Adware*. Blog Post. 2015.

[Appd] *Content Providers*. `https://developer.android.com/guide/topics/providers/content-providers.html`. last checked: Oct. 2016.

[Coo+09] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. "Automatic Static Unpacking of Malware Binaries." In: *Proceedings of the 16th Working Conference on Reverse Engineering*. IEEE Computer Society, 2009.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08. Springer-Verlag, 2008.

[Deva] Android Developers. *Android NDK*.

[Devb] Android Developers. *Settigs.Secure*.

[Dex] DexProtector. *DexProtector User Manual*.

[DT05] Rachna Dhamija and J. D. Tygar. "The Battle Against Phishing: Dynamic Security Skins." In: *Proceedings of the 2005 Symposium on Usable Privacy and Security*. SOUPS '05. ACM, 2005.

[Do+16] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. *Just-in-Time Static Analysis*. Tech. rep. University of Alberta, Aug. 2016.

[Dra+14] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android hacker's handbook*. John Wiley & Sons, 2014.

[Dun+14] Ken Dunham, Shane Hartman, Manu Quintans, Jose Andre Morales, and Tim Strazzere. *Android Malware and Analysis*. 1st. Auerbach Publications, 2014.

[Ege+13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. "An Empirical Study of Cryptographic Misuse in Android Applications." In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. ACM, 2013.

[Enc+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. USENIX Association, 2010.

[Enc+11a] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. "A Study of Android Application Security." In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. USENIX Association, 2011.

[Enc+11b] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. "A study of android application security." In: *Proceedings of the 20th USENIX conference on Security*. SEC'11. USENIX Association, 2011.

[Eri16] Ericsson. *Ericsson Mobility Report*. `http://www.ericsson.com/res/docs/2016/mobility-report/ericsson-mobility-report-feb-2016-interim.pdf`. Report. 2016.

[FS14] F-Secure. *Mobile Threat Report Q1 2014*. `http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2014_print.pdf`. Apr. 2014.

[Fel+11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. "Android permissions demystified." In: *Proceedings of the 18th ACM conference on Computer and communications security*. CCS '11. ACM, 2011.

[For] *Fortify 360 Source Code Analyzer (SCA)*. `http://www8.hp.com/us/en/software-solutions/software.html?compURI=1214365#.UW6CVKuAtfQ`. Apr. 2013.

[Fra+15a] Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. "CLAPP: Characterizing Loops in Android Applications." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. ACM, 2015.

[Fra+15b] Yanick Fratantonio, Antonio Bianchi, William Robertson, Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. "On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users." In: *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. 2015.

[Fra+16] Yanick Fratantonio, Antonio Bianchi, William K. Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. "TriggerScope: Towards Detecting Logic Bombs in Android Applications." In: *Proceedings of the Symposium on Security and Privacy (SP)*. IEEE, 2016.

[FCF09] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. *SCanDroid: Automated Security Certification of Android Applications*. Tech. rep. CS-TR-4991. Department of Computer Science, University of Maryland, College Park, 2009.

[Fut15] IDC Analyze the Future. *Smartphone OS Markte Share, 2015 Q2*. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. Online. 2015.

[Gib+12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. "AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale." In: *Proceedings of the 5th international conference on Trust and Trustworthy Computing*. TRUST'12. Springer-Verlag, 2012.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. ACM, 2005.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2008.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing." In: *Queue* 10.1 (Jan. 2012).

[Chr] *Google Cast*. `https://developers.google.com/cast`. 2013.

[Goo14a] Google Developers. *monkeyrunner*. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`. Google Developer Website. 2014.

[Goo] *Google Glass*. `https://developers.google.com/glass/`. 2013.

[Goo14b] Google Play. *WhatsApp Messenger*. Google PlayStore Website. `https://play.google.com/store/apps/details?id=com.whatsapp`. 2014.

[Gor+15] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. "Information-Flow Analysis of Android Applications in DroidSafe." In: *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*. 2015.

[Gor+14] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. "Checking App Behavior Against App Descriptions." In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. 2014.

[Gra+12] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. "Unsafe Exposure Analysis of Mobile In-app Advertisements." In: *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WISEC '12. ACM, 2012.

[Hac16] Julien Hachenberger. "Detecting Android Obfuscation Techniques." Master Thesis. Hochschule Darmstadt, Nov. 2016.

[Hal+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. "The WEKA data mining software: an update." In: *ACM SIGKDD Explorations Newsletter* 11 (2009).

[Hao+14] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps." In: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '14. ACM, 2014.

[Has+15] Behnaz Hassanshahi, Yaoqi Jia, Roland H. C. Yap, Prateek Saxena, and Zhenkai Liang. "Web-to-Application Injection Attacks on Android: Characterization and Detection." In: *Proceedings of the 20th European Symposium on Research in Computer Security*. Springer International Publishing, 2015.

[HP00] Klaus Havelund and Thomas Pressburger. "Model checking JAVA programs using JAVA PathFinder." In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000).

[He14] Dongjing He. "Security Threats to Android Apps." MA thesis. University of Illinois at Urbana-Champaign, 2014.

[Ida] *Hex-Rays: IDAPro*. https://www.hex-rays.com/products/ida/. 2016.

[Hof14] Johannes Hoffmann. "From Mobile to Security - Towards Secure Smartphones." PhD thesis. Ruhr-Universität Bochum, 2014.

[Hof+13] Johannes Hoffmann, Martin Ussath, Michael Spreitzenbarth, and Thorsten Holz. "Slicing Droids: Program Slicing for Smali Code." In: *Proceedings of the 28th Symposium On Applied Computing*. Ed. by ACM. 2013.

[HN11] Cuixiong Hu and Iulian Neamtiu. "Automating GUI Testing for Android Applications." In: *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011.

[Hu+14] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. "Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor." In: *EuroSys* (2014).

[Appe] *IBM Rational AppScan*. http://www-01.ibm.com/software/de/rational/appscan/. Apr. 2013.

[Appf] *Intents and Intent Filters*. https://developer.android.com/guide/components/intents-filters.html. last checked: Oct. 2016.

[Jeb] *JEB Decompiler*. https://www.pnfsoftware.com/. 2016.

[JPM13a] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. "Automated Testing with Targeted Event Sequence Generation." In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA '13. ACM, 2013.

[JPM13b] Casper Svenning Jensen, Mukul R. Prasad, and Anders Møller. "Automated testing with targeted event sequence generation." In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. ACM, 2013.

[Jeo+12] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. "Dr. Android and Mr. Hide: fine-grained permissions in android applications." In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. SPSM '12. ACM, 2012.

[JGM15] Bahman Pedrood Ali Bagheri-Khaligh Joshua Garcia Mahmoud Hammad and Sam Malek. *Obfuscation-Resilient, Efficient, and Accurate Detection and Family Identification of Android Malware*. Tech. rep. Department of Computer Science, George Mason University, 2015.

[Kim+12] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. "ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications." In: *MoST 2012: Mobile Security Technologies*. IEEE, May 2012.

[Kin+08] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. "Implicit Flows: Can't Live with 'Em, Can't Live without 'Em." In: *Proceedings of the 4th International Conference on Information Systems Security*. ICISS '08. Springer Berlin Heidelberg, 2008.

[Kin76] J. C. King. "Symbolic Execution and Program Testing." In: *Communications of the ACM* (1976).

[Kli+14] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. "Android Taint Flow Analysis for App Sets." In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. SOAP '14. ACM, 2014.

[Koh95] Ron Kohavi. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection." In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. IJCAI'95. Morgan Kaufmann, 1995.

[Kol+12a] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. "Rozzle: De-cloaking Internet Malware." In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. IEEE Computer Society, 2012.

[Kol+12b] Clemens Kolbitsch, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. "Rozzle: De-cloaking Internet Malware." In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. 2012.

[LLC15] Pulse Secure LLC. *Mobile Threat Report 2015*. Whitepaper. 2015.

[Lab14] International Secure Systems Lab. *Anubis - Malware Analysis for Unknown Binaries*. `http://anubis.iseclab.org`. Anubis Website. 2014.

[Lam+11] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. "The Soot framework for Java program analysis: a retrospective." In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. 2011.

[Li+14] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. "Automatically Exploiting Potential Component Leaks in Android Applications." In: *Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. TRUSTCOM '14. IEEE Computer Society, 2014.

[Li+15a] Li Li, Alexandre Bartel, Tegawende Bissyande, Jacques Yves Klein, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps." In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE 2015. ACM, 2015.

[Li+15b] Li Li, Alexandre Bartel, Tegawende F. Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps." In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE '15. 2015.

[Li+15c] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F. Bissyandé, and Jacques Klein. "Potential Component Leaks in Android Apps: An Investigation into a New Feature Set for Malware Detection." In: *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*. QRS '15. 2015.

[Li+16a] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. "DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps." In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. 2016.

[Li+16b] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. "Reflection-Aware Static Analysis of Android Apps." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. 2016.

[Li13] Zhe Li. "Fake KakaoTalk security plug-in." In: *Virus Bulletin* (2013).

[LY16] Jongsu Lim and Jeong Hyun Yi. "Structural analysis of packing schemes for extracting hidden codes in mobile malware." In: *EURASIP Journal on Wireless Communications and Networking* 2016 (2016).

[Lin+14] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors." In: *Proceedings of the the 3rd International Workshop BADGERS*. 2014.

[Liu+15] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. "Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps." In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '15. ACM, 2015.

[Liv+09] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. "Merlin: Specification Inference for Explicit Information Flow Problems." In: *SIGPLAN Not.* 44 (June 2009).

[Lu+12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. "CHEX: statically vetting Android apps for component hijacking vulnerabilities." In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS '12. ACM, 2012.

[MTN13] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: An Input Generation System for Android Apps." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. ACM, 2013.

[MMM14] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. "EvoDroid: Segmented Evolutionary Testing of Android Apps." In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. ACM, 2014.

[MMP14] D. Maier, T. Muller, and M. Protsenko. "Divide-and-Conquer: Why Android Malware Cannot Be Stopped." In: *Proceedings of the 2014 Ninth International Conference on Availability, Reliability and Security*. ARES '14. 2014.

[MF11] Jan Malburg and Gordon Fraser. "Combining Search-based and Constraint-based Testing." In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. IEEE Computer Society, 2011.

[MS12] Christopher Mann and Artem Starostin. "A framework for static detection of privacy leaks in android applications." In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. ACM, 2012.

[Mei12] Reto Meier. *Professional Android 4 application development*. John Wiley & Sons, 2012.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities." In: *Commun. ACM* (Dec. 1990).

[Mir+12] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. "Testing Android Apps Through Symbolic Execution." In: *SIGSOFT Softw. Eng. Notes* (Nov. 2012).

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Exploring Multiple Execution Paths for Malware Analysis." In: *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. SP '07. IEEE Computer Society, 2007.

[Nak14] Daisuke Nakajima. *Vietnamese Adult Apps on Google Play Open Gate to SMS Trojans*. `http://blogs.mcafee.com/mcafee-labs/vietnamese-adult-apps-google-play-open-gate-to-sms-trojan`. McAfee Labs Website. 2014.

[OM12a] J Oberheide and C Miller. "Dissecting the android bouncer." In: *SummerCon2012, New York* (2012).

[OM12b] Jon Oberheide and Charlie Miller. *Dissecting the Android Bouncer*. Talk at Summercon 2012. 2012.

[OJM12] Damien Octeau, Somesh Jha, and Patrick McDaniel. "Retargeting Android Applications to Java Bytecode." In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. ACM, 2012.

[Oct+13] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. "Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis." In: *Proceedings of the USENIX Security '13*. USENIX Association, 2013.

[Oct+15] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis." In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. 2015.

[Oct+16] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. "Combining Static Analysis with Probabilistic Models to Enable Market-scale Android Inter-component Analysis." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. ACM, 2016.

[Ora] Oracle. *Invoking Methods*. `https://docs.oracle.com/javase/tutorial/reflect/member/methodInvocation.html`.

[Pen+14] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. "X-force: Force-executing binary programs for security applications." In: *Proceedings of the 2014 USENIX Security Symposium, San Diego, CA (August 2014)*. 2014.

[Pen+12] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. "Using probabilistic generative models for ranking risks of Android apps." In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS '12. ACM, 2012.

[Pet+14] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. "Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware." In: *EuroSec '14*. ACM, 2014.

[PO12] H. Pieterse and M. S. Olivier. "Android botnets on the rise: Trends and characteristics." In: *Proceedings of the 2012 Information Security for South Africa*. 2012.

[Andd] *Platform Architecture*. `https://developer.android.com/guide/platform/index.html`. last checked: Oct. 2016.

[Pla98] John C. Platt. "Fast training of support vector machines using sequential minimal optimization." In: *Advances in Kernel Methods – Support Vector Learning*. MIT Press, 1998.

[PB15] Andrey Polkovnichenko and Alon Boxiner. *BrainTest – A New Level of Sophistication in Mobile Malware*. Blog Post. 2015.

[PK16] Andrey Polkovnichenko and Oren Koriat. *Viking Horde: A New Type of Android Malware on Google Play*. Blog Post. 2016.

[Qui93] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[RKK07] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. "Detecting System Emulators." In: *Proceedings of the 10th International Conference on Information Security*. ISC '07. Springer-Verlag, 2007.

[RA15] Siegfried Rasthofer and Steven Arzt. "(In-)Security of Backend-As-A-Service Solutions." In: *Blackhat Europe*. Nov. 2015.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks." In: *2014 Network and Distributed System Security Symposium (NDSS)*. 2014.

[RCH15] Siegfried Rasthofer, Carlos Castillo, and Alex Hichliffe. "We know what you did this Summer: Android Banking Trojan Exposing its Sins in the Cloud." In: *18th Association of Anti-virus Asia Researchers International Conference (AVAR) 2015*. Dec. 2015.

[Ras+14] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. "DroidForce: Enforcing Complex, Data-Centric, System-Wide Policies in Android." In: *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*. IEEE. Sept. 2014.

[Ras+15a] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. *An Investigation of the Android/BadAccents Malware which Exploits a new Android Tapjacking Attack*. Tech. rep. TU Darmstadt and McAfee Research Lab, Apr. 2015.

[Ras+15b] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. "How Current Android Malware Seeks to Evade Automated Code Analysis." In: *9th International Conference on Information Security Theory and Practice (WISTP'2015)*. 2015.

[Ras+16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques." In: *2016 Network and Distributed System Security Symposium (NDSS)*. San Diego, 2016.

[RCE13] Vaibhav Rastogi, Yan Chen, and William Enck. "AppsPlayground: Automatic Security Analysis of Smartphone Applications." In: *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*. CODASPY '13. ACM, 2013.

[RC15] Idan Revivo and Ofer Caspi. "CukooDroid - An Automated Malware Analysis Framework." In: *Blackhat USA*. Aug. 2015.

[Rie+11] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. "Automatic Analysis of Malware Behavior Using Machine Learning." In: *J. Comput. Secur.* (Dec. 2011).

[RM10] Kevin A. Roundy and Barton P. Miller. "Hybrid Analysis and Control of Malware." In: *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*. RAID'10. Springer-Verlag, 2010.

[Rub+16] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. "Automated Partitioning of Android Applications for Trusted Execution Environments." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. ACM, 2016.

[Rui12] Fernando Ruiz. *FakeInstaller Leads the Attack on Android Phones*. `https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones`. McAfee Labs Website. 2012.

[Rui16] Fernando Ruiz. *Android Malware Clicker.G!Gen Found on Google Play*. Blog Post. 2016.

[Sar+12] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. "Android permissions: a perspective combining risks and benefits." In: *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*. SACMAT '12. ACM, 2012.

[Sax+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. "A Symbolic Execution Framework for JavaScript." In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. IEEE Computer Society, 2010.

[Sch+13] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. "Dynamic Determinacy Analysis." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. ACM, 2013.

[Sch03] Karl-Michael Schneider. "A comparison of event models for Naive Bayes anti-spam e-mail filtering." In: *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1*. EACL '03. Association for Computational Linguistics, 2003.

[Seb+02] Abdallah Abbey Sebyala, Temitope Olukemi, Lionel Sacks, and Dr. Lionel Sacks. "Active Platform Security through Intrusion Detection Using Naive Bayesian Network For Anomaly Detection." In: *Proceedings of London communications symposium*. 2002.

[Sen07] Koushik Sen. "Concolic Testing." In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. ACM, 2007.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. ACM, 2005.

[Sha+12] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. ""Andromaly": a behavioral malware detection framework for android devices." In: *Journal of Intelligent Information Systems* 38 (2012).

[Sla+16] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. "Toward a Framework for Detecting Privacy Policy Violations in Android Application Code." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. ACM, 2016.

[Sne16] Bruce Snell. *Mobile Threat Report: What's on the Horizon for 2016*. `http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf`. 2016.

[Sou+14] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. "SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2014.

[Spr+13] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. "Mobile-sandbox: Having a Deeper Look into Android Applications." In: *Proceedings of the 28th Annual ACM SAC*. SAC '13. ACM, 2013.

[Sta14] AppBrain Stats. *Number of Android applications*. `http://www.appbrain.com/stats/number-of-android-apps`. Android Statistics Page of AppBrain. 2014.

[Ande] *System Permissions*. `https://developer.android.com/guide/topics/security/permissions.html`. last checked: Oct. 2016.

[Szo05] Peter Szor. *The art of computer virus research and defense*. Pearson Education, 2005.

[Tam+15] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors." In: *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[Tec14] Saikoa Applied Compiler Technology. *DexGuard*. `http://www.saikoa.com/dexguard`. Saikoa Website. 2014.

[Geo] *The Google Maps Geolocation API*. `https://developers.google.com/maps/documentation/business/geolocation/`. 2013.

[Thu+11] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. "Synthesizing method sequences for high-coverage testing." In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. 2011.

[Tia16] Di Tian. *Detecting vulnerabilities of broadcast receivers in Android applications*. 2016.

[TKG13] Emre Tinaztepe, Doğan Kurt, and Alp Güleç. *Android OBAD*. Tech. rep. COMODO, July 2013.

[Tip94] Frank Tip. *A Survey of Program Slicing Techniques*. Tech. rep. 1994.

[TS15] Dennis Titze and Julian Schütte. "Apparecium: Revealing data flows in android applications." In: *Proceedings of the 29th International Conference on Advanced Information Networking and Applications*. IEEE. 2015.

[Tri+16] Omer Tripp, Marco Pistoia, Pietro Ferrara, and Julia Rubin. "Pinpointing Mobile Malware Using Code Analysis." In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. MOBILESoft '16. ACM, 2016.

[Andf] *Understand the APK Structure*. `https://developer.android.com/topic/performance/reduce-apk-size.html`. last checked: Oct. 2016.

[VR+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. "Soot - a Java bytecode optimization framework." In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. IBM, 1999.

[VC14a] T. Vidas and N. Christin. "Evading Android Runtime Analysis via Sandbox Detection." In: *ASIACCS '14*. June 2014.

[VC14b] Timothy Vidas and Nicolas Christin. "Evading Android Runtime Analysis via Sandbox Detection." In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. ACM, 2014.

[Vir] *Virus Share*. `http://virusshare.com/`. 2013.

[Web16] Dr. Web. *More than 100 Google Play apps found to contain advertising spyware*. Blog Post. 2016.

[Wei+14] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. ACM, 2014.

[Wei81] Mark Weiser. "Program Slicing." In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. IEEE Press, 1981.

[WL16] Michelle Y. Wong and David Lie. "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2016.

[Xpo] *Xposed*. `http://repo.xposed.info/`. June 2016.

[XSA12] Rubin Xu, Hassen Saïdi, and Ross Anderson. "Aurasium: practical policy enforcement for Android applications." In: *Proceedings of the 21st USENIX conference on Security symposium*. Security'12. USENIX Association, 2012.

[Xu+14] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. "GoldenEye: Efficiently and Effectively Unveiling Malware's Targeted Environment." In: *Proceedings of the 17th International Symposion, Research in Attacks, Intrusions and Defenses*. Lecture Notes in Computer Science. Springer International Publishing, 2014.

[YZA08] Wei Yan, Zheng Zhang, and Nirwan Ansari. "Revealing packed malware." In: *IEEE Security & Privacy* 6.5 (2008).

[YY12] Zhemin Yang and Min Yang. "LeakMiner: Detect Information Leakage on Android with Static Taint Analysis." In: *Third World Congress on Software Engineering (WCSE 2012)*. 2012.

[Yan+13] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. "AppIntent: analyzing sensitive data transmission in android for privacy leakage detection." In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. CCS '13. ACM, 2013.

[Yin+16] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. "Attacks and Defence on Android Free Floating Windows." In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. ACM, 2016.

[Yu+15] Le Yu, Tao Zhang, Xiapu Luo, and Lei Xue. "AutoPPG: Towards Automatic Generation of Privacy Policy for Android Applications." In: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '15. ACM, 2015.

[Zha04] Harry Zhang. "The Optimality of Naive Bayes." In: *FLAIRS Conference*. Ed. by Valerie Barr and Zdravko Markov. AAAI Press, 2004.

[ZY14] Mu Zhang and Heng Yin. "AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Feb. 2014.

[ZO12] Zhibo Zhao and Fernando C. Colón Osorio. ""TrustDroid;": Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking." In: *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software*. MALWARE '12. 2012.

[ZAH11] Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. "Automatic Extraction of Secrets from Malware." In: *Proceedings of the 18th Working Conference on Reverse Engineering*. IEEE Computer Society, 2011.

[ZSL13] Min Zheng, Mingshen Sun, and John C. S. Lui. "DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware." In: *CoRR* abs/1302.7212 (2013).

[ZZG13] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. "Z3-str: A Z3-based String Solver for Web Application Analysis." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. ACM, 2013.

[ZJ12] Yajin Zhou and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution." In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. IEEE Computer Society, 2012.

[Zho+15] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. "Harvesting Developer Credentials in Android Apps." In: *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '15. 2015.

[Cya] *cyanogenmod*. `http://www.cyanogenmod.org/`. Apr. 2013.

[Jad] *jadx - Dex to Java decompiler*. `https://github.com/skylot/jadx`. Github project. 2016.

[sec16] info security. *Pincer.A – new Android trojan warning*. `http://www.infosecurity-magazine.com/news/pincera-new-android-trojan-warning/`. last visited: November 2016. 2016.

[Sma] *smalidea*. `https://github.com/JesusFreke/smali/wiki/smalidea`. Github project. 2016.

Erklärung gemäß §9 der Promotionsordnung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Deutschland, Januar 2017

Siegfried Rasthofer