PhD-FSTM-2023-011
The Faculty of Science, Technology and Medicine

Dissertation

Defense held on 09/01/2023 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Jordan Samhi

Born on the 27^{th} of October, 1991 in Metz, France

Analyzing the Unanalyzable: an Application to Android Apps

Dissertation Defense Committee

Dr. Jacques Klein, Dissertation Supervisor
Full Professor, University of Luxembourg

Dr. Tegawendé F. Bissyandé, Chairman
Associate Professor, University of Luxembourg

Dr. Michael D. Ernst, Vice-Chairman
Full Professor, University of Washington

Dr. Andreas Zeller
Full Professor, CISPA Helmholtz Center for Information Security, Saarland University

Dr. Mauro Conti
Full Professor, University of Padua

Abstract

In general, software is unreliable. Its behavior can deviate from users' expectations because of bugs, vulnerabilities, or even malicious code. Manually vetting software is a challenging, tedious, and highly-costly task that does not scale. To alleviate excessive costs and analysts' burdens, automated static analysis techniques have been proposed by both the research and practitioner communities making static analysis a central topic in software engineering. In the meantime, mobile apps have considerably grown in importance. Today, most humans carry software in their pockets, with the Android operating system leading the market. Millions of apps have been proposed to the public so far, targeting a wide range of activities such as games, health, banking, GPS, etc. Hence, Android apps collect and manipulate a considerable amount of sensitive information, which puts users' security and privacy at risk. Consequently, it is paramount to ensure that apps distributed through public channels (e.g., the Google Play) are free from malicious code. Hence, the research and practitioner communities have put much effort into devising new automated techniques to vet Android apps against malicious activities over the last decade.

Analyzing Android apps is, however, challenging. On the one hand, the Android framework proposes constructs that can be used to evade dynamic analysis by triggering the malicious code only under certain circumstances, e.g., if the device is not an emulator and is currently connected to power. Hence, dynamic analyses can -easily- be fooled by malicious developers by making some code fragments difficult to reach. On the other hand, static analyses are challenged by Android-specific constructs that limit the coverage of off-the-shell static analyzers. The research community has already addressed some of these constructs, including inter-component communication or lifecycle methods. However, other constructs, such as implicit calls (i.e., when the Android framework asynchronously triggers a method in the app code), make some app code fragments unreachable to the static analyzers, while these fragments are executed when the app is run. Altogether, many apps' code parts are unanalyzable: they are either not reachable by dynamic analyses or not covered by static analyzers.

In this manuscript, we describe our contributions to the research effort from two angles: ① statically detecting malicious code that is difficult to access to dynamic analyzers because they are triggered under specific circumstances; and ② statically analyzing code not accessible to existing static analyzers to improve the comprehensiveness of app analyses. More precisely, in Part I, we first present a replication study of a state-of-the-art static logic bomb detector to better show its limitations. We then introduce a novel hybrid approach for detecting suspicious hidden sensitive operations towards triaging logic bombs. We finally detail the construction of a dataset of Android apps automatically infected with logic bombs. In Part II, we present our work to improve the comprehensiveness of Android apps' static analysis. More specifically, we first show how we contributed to account for atypical inter-component communication in Android apps. Then, we present a novel approach to unify both the bytecode and native in Android apps to account for the multi-language trend in app development. Finally, we present our work to resolve conditional implicit calls in Android apps to improve static and dynamic analyzers.

Acknowledgement

I want to express my most profound appreciation to the people who made this dissertation possible. Many people have contributed to this journey directly by extending their precious knowledge, advice, and experience; or indirectly by supporting me daily.

Firstly, I would like to express my deepest gratitude to my supervisor, Prof. Jacques Klein, who trusted me and gave me the opportunity to pursue a doctoral degree with renowned researchers. This endeavor would not have been possible without his support and trust throughout my Ph.D. journey.

Secondly, I am also deeply thankful to my daily advisor, Prof. Tegawendé F. Bissyandé, who gave me invaluable advice and discussions on soft skills management which helped me develop a panel of aptitudes. I am particularly grateful to Prof. Jacques Klein and Prof. Tegawendé F. Bissyandé for their patience and valuable guidance. In particular, they have taught me to perform research, write technical papers, develop critical thinking w.r.t. to research articles, and prepare and conduct engaging presentations. Besides, thanks to their openness, I learned how to manage students and interns. I had the opportunity to teach, participated in many research community activities, and learned how to write a research project, which prepared me for my future career. I cannot thank them enough.

Thirdly, I profoundly thank Prof. Michael D. Ernst and Prof. René Just, who welcomed me to the University of Washington for a couple of months and gave me precious advice and another picture of research.

Fourthly, I would like to extend my sincere thanks to all my co-authors, including Prof. Alexandre Bartel, Prof. Li Li, Dr. Steven Arzt, Dr. Kevin Allix, Dr. Pierre Graux, Dr. Jun Gao, Dr. Pingfan Kong, Nadia Daoudi, Abdoul Kader Kabore, Maria Kober, Henri Hoyez, and Xiaoyu Sun for their helpful conversations and collaborations.

I want to thank the members of my Ph.D. defense committee, including chairman Prof. Tegawendé F. Bissyandé, vice-chairman Prof. Michael D. Ernst, Prof. Andreas Zeller, Prof. Mauro Conti, Prof. Li Li, and my supervisor Prof. Jacques Klein. It is my great honor to have such a preeminent defense committee. I appreciate and recognize their effort to examine my manuscript and evaluate my Ph.D. work.

With great pleasure, I express my gratitude to all TruX research group team members and the University of Luxembourg administrative team.

Words cannot express my gratitude to my wife, Anaïs, for her love, her constant support, and for making me who I am. Thank you for believing in me. I could not have undertaken this journey without the support over the years of my family, i.e., Claude, Laurence, and Émeline. My brother, Thomas, deserves thanks for his thorough and persistent support and for allowing me to release the pressure when I needed it the most.

Jordan Samhi
University of Luxembourg
December 2022

To my dearest wife Anaïs.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1

Introduction

In this chapter, we first introduce the problems addressed by our research. Then, we summarize the challenges analysts face when analyzing Android applications. Finally, we succinctly present the contributions of our work and the roadmap of this dissertation.

Contents

1.1 Problem Statement

Software is ubiquitous. Security and privacy threats are growing with its integration into various commodity devices. Thus, avoiding critical consequences on software systems is a prime concern for researchers and practitioners. Nevertheless, given the gigantic number of software available in various domains, human analysts' work cannot scale to ensure more secure and reliable code. Hence, automating code analysis is key, as stated by Bryan Palma, CEO of McAfee/FireEye, in 2021: "*automation is the only way forward for cybersecurity*" [1]. That is why researchers and practitioners have investigated numerous techniques to vet software both without executing it, i.e., static analysis [2, 3, 4, 5, 6, 7, 8, 9, 10, 11], and during execution, i.e., dynamic analysis [12, 13, 14, 15, 16]. However, the aforementioned techniques suffer from a lack of holistic view, which hinders comprehensive analyses.

Our work focus on the Android ecosystem. Therefore, static and dynamic analyzers are transitively challenged by specific mechanisms provided by the Android framework to build Android applications (apps). Indeed, apps are mainly built upon the *callback* paradigm, which impacts analyses' efficiency for the following reasons. First, beyond the strong constraints to generate correct inputs to reach parts of the code, Android dynamic analyzers also need to simulate callback-related tasks such as button clicks to ensure acceptable coverage. Also, many Android constructs allow to differ code execution based on specific conditions [17] (e.g., the network needs to be connected, the device is in charge, execution in 1 hour, etc.). Hence, dynamic analyzers would not cover parts of the code if the conditions are not met. Second, the Android framework acts as a black box for static analyzers. Indeed, static analysis cannot afford, for scaling issues, to analyze the Android framework while analyzing app code. Besides, callback-related behaviors (e.g., inter-component analysis, job-related tasks) work at the framework level. Therefore, many discontinuities are found in Android apps between method calls and the actual behavior triggered [18, 8, 6] (e.g., a call to method m in the app code that would trigger method n in the app code, see Figure 1.1). These discontinuities lead to parts of the app code being left out by static analyzers since they cannot "see" them. For instance, in Figure 1.1, if method n is never called in the app code, it would not be analyzed by a static analyzer performing a data flow analysis.

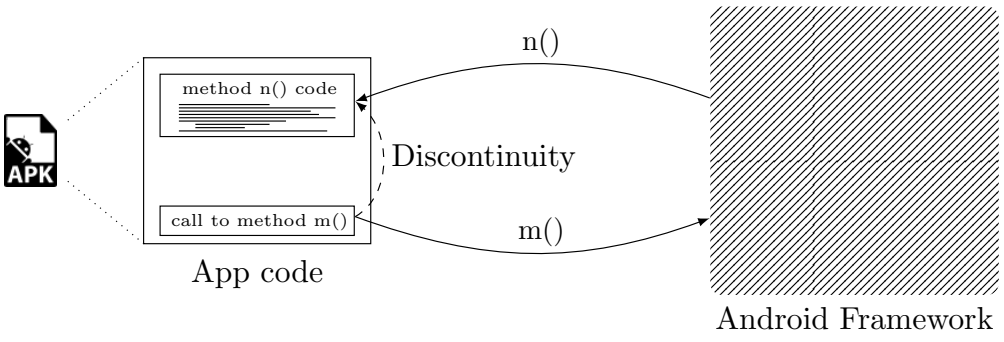


Figure 1.1: An example of a discontinuity in an Android app code.

Besides, the malware industry is continually rising, especially in the Android ecosystem, since roughly three-quarters of people owning a mobile device rely on the Android operating system, thus there is 50 times more Android malware than iOS malware [19]. In 2022, so far, more than 6.4 million Android malware have been found by security analysts and anti-virus companies with an 80% increase of banking malware threats [19]. However, it has been shown that 47% of free anti-viruses cannot detect malware effectively. Hence, Android-based devices are highly at risk. Furthermore, nowadays, malicious developers

put a lot of effort into hiding the malicious from dynamic analyzers, e.g., making the code dormant and executing it only under certain circumstances. For instance, the *Mandrake* malware managed to stay hidden for years before being detected as malware [20]. More recently, researchers from the Satori Threat Intelligence and Research Team found an umpteenth version of the Poseidon malware that relies on conditional implicit calls to start the malicious process [21]. Hence, and as described above, the discontinuity between the app code and the implicit call makes static analyzers miss the most important code, i.e., the malicious code. Consequently, malicious applications are still entering the official Google app market, i.e., the Google Play in 2022.

Although manually vetting Android apps is one of the most effective techniques to detect malicious code, it is a challenging and extremely highly-costing process, which app market providers cannot afford to avoid bottlenecks. To reduce excessive costs and analysts' burdens, automated techniques must be devised. The researcher and practitioner communities have already proposed static and dynamic techniques. However, recent events show that malicious developers still enter the Google Play [22, 23] using the current limitations of existing approaches.

Static analysis holds the promise of analyzing code without executing it, and even covering code that might not be executed at runtime (e.g., hidden code). In a software development life cycle (SDLC) environment, static analysis is used to detect programming errors, vulnerabilities, bugs, stylistic errors, and erroneous constructs. Anti-virus companies and app market providers rely on static analysis to detect malicious code. Though static analysis has proven to be effective, there still exist many areas of improvement to be explored.

In this dissertation, we propose to develop new static code analysis techniques for two purposes: ① to expose code that is unreachable for dynamic analyzers, e.g., logic bombs; and ② to expose code that is unreachable for existing static analyzers, i.e., improving the comprehensiveness of software analysis.

1.2 Challenges

In this section, we describe the technical challenges analysts face when they statically analyze Android applications. More specifically, we describe these challenges from 6 points of view: ① challenges related to Android app modeling; ② challenges related to scaling issues; ③ challenges related to the high amount of false positive generated by static analyzers; ④ challenges related to the detection of malicious code; ⑤ challenges related to the multi-language trend in Android apps; and ⑥ challenges related to obfuscation.

1.2.1 Android Applications Modeling

Contrary to traditional Java programs, Android apps do not have a single entry point, i.e., a main function that is invoked when the program is executed and from which the rest of the execution follows. Rather, Android apps can be viewed as a constellation of components that implicitly interact with the Android framework. Indeed, the notion of *component* is central in Android app development. The Android framework provides four types of components: ① the `Activity` component that implements the UI visible to users; ② `Service` components run background tasks; ③ `Content Provider` components expose shared databases; and ④ `Broadcast Receiver` are components triggered by system events. Developers need to implement Java classes (or Kotlin classes) that directly inherit one of these components to use them. Furthermore, these components provide several callback methods that are never called within the app itself but triggered implicitly by the Android framework according to the current state of the app's lifecycle. An example of an Activity component's lifecycle state diagram is visible in Figure 1.2. These different states allow users to easily interact with other apps on the device, e.g., switching between apps would pause the foreground apps and activate the next apps. The relevant callback methods are directly managed internally by the Android framework.

In addition, since Android apps are made to interact with users, e.g., via buttons, Android apps are mainly callback driven. Indeed, the Android runtime is constantly polling for user input to trigger appropriate methods defined by the app developer, e.g., a particular button. Therefore, likewise lifecycle methods, there is no explicit call to callback methods in apps since this is handled by the Android framework.

This constitutes the first challenge related to statically analyzing Android apps. Indeed, since there is no single entry point and there is no explicit call to lifecycle callback methods in the app itself, and considering that static analyzers need an entry point where to start an analysis, and need explicit calls to construct call graphs, analysts need to find the proper solution to handle this challenge. Previous work provides a solution to this problem. Indeed, FlowDroid [5] proposed an instrumentation technique to provide a single entry point to a given app. This entry point is called the *dummy main method*. In addition to that, FlowDroid creates a component *dummy method* which sequentially and explicitly calls the lifecycle methods one by one. In turn, the app *dummy main method* itself sequentially and explicitly calls the components *dummy method*. Both "fake" methods would protect the execution of, normally implicit methods, with opaque predicates in order to allow static analyzers to equally consider all possible paths, in order to simulate the uncertainty with which lifecycle method and callback methods would be triggered at run time.

However, Android apps and the Android framework require more attention and advanced digging. Indeed, implicit calls are pervasive in Android apps, though not always documented properly. The literature already provides solutions for a couple of implicit mechanisms, e.g., IccTA [6] for inter-component communication, DroidRA [8] for reflection, etc. In this dissertation, we show that other implicit call mechanisms have been overlooked by the state of the art. These mechanisms need special attention since, as

we will show, they are used by malware writers. Consequently, if static analyzers cannot properly model and represent at best Android apps and their control flow, the analysis might not be comprehensive, and, e.g., malicious code might be missed. In Chapter 14 we describe why a more thorough and systematic study of the Android framework itself is needed as future work to better understand the communication and implicit relations between apps and the framework.

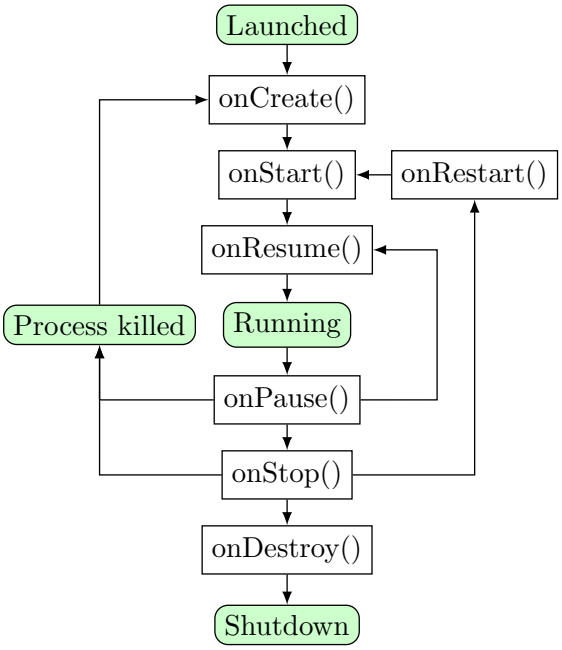


Figure 1.2: Activity component life cycle

1.2.2 Scaling Challenges

Automated analyses are limited in terms of memory and CPU time. Static analysis is no exception to the rule. Besides, the Android ecosystem puts more constraints on static analyzers, which further hinder scalability. Indeed, developers make thorough use of polymorphism, which implies over-approximation from static analyzers. Over-approximation is one of the reasons static analyzers do not scale, in general, since it makes explode both: ① the number of edges in the call graph for a given app, which heavily increases the number of paths to explore; and ② the number of dataflow values to be stored for different contexts. Besides, with the concept of reusability, many libraries are used in Android apps, and with hundreds of thousands of methods in the Android framework, the number of paths to consider by static analyzers is immense. All these concepts and artifacts would make static analyzers to be executed for months, years, or even more and could rapidly saturate the memory available. Hence, it is common in the static analysis world to: ① avoid analyzing library code; ② avoid analyzing the Android framework code; and ③ rely on timeouts to avoid infinite execution. Consequently, most of the time, static analyzers over-approximate results and might miss relevant results.

1.2.3 False Positives

Static analysis is noisy. Indeed, a common characteristic of all prototypes is that they sometimes report incorrect results, i.e., false positives. As aforementioned, static analyzers have to make assumptions about the expected behavior of the code, i.e., they often *over-approximate*. This inherently generates false alarm results since the results yielded might

not be representative of the expected run time behavior. However, certain tools cause more false positives than others, given the set of rules used to detect a property (i.e., more or fewer constraints). Therefore, it is a common practice in the static analysis community to manually examine tools' output to assess their efficiency on real-world samples.

1.2.4 Malicious Code Detection

What is a piece of malicious code? How to characterize it statically? These questions are part of the open problem of trying to detect malicious code in software, which is undecidable in the general case. A piece of malicious code in a specific app might not be considered as malicious in another. For instance, an app that would collect users' real-time location might be considered suspicious/malicious in a calculator, but maybe not in a GPS app. Hence, statically and automatically characterizing pieces of malicious code is challenging for analysts. However, machine-learning techniques have proven to be efficient in classifying apps as benign or malicious in the last decade.

1.2.5 Multi-Language Trend

Android apps are mainly developed in either Java or Kotlin programming languages. Both can be compiled to produce Dalvik bytecode that runs over the Dalvik virtual machine embedded in the Android operating system. However, many other options are available to developers for providing end-users with functionalities. Indeed, for instance, the Android Native Development Kit (NDK) allows developers to incorporate C and C++ code in Android apps. Developers usually prefer these languages to perform CPU-intensive tasks since it executes faster than a virtual-machine-based language. Another prominent example is the use of more and more JavaScript in Android's Webviews which allows for rich user interfaces and experience. Other languages can be used within Android apps, such as Python, C#, LUA, etc. The literature has so far mainly focused on analyzing the Dalvik bytecode available in Android apps. Few works tried to account for native code and proposed approaches to analyze it independently from the bytecode. Therefore, there is a gap in current Android apps and the way they are represented by static analyzers that do not account for languages other than the Dalvik bytecode. Consequently, there is an urge in the community to propose new approaches that account for all possible pieces of code that make an app towards comprehensive analyses.

1.2.6 Obfuscation

Obfuscation is a process of making code difficult to read for both humans and machines. It is mainly used for two purposes poles apart: ① many companies rely on obfuscation to protect their intellectual property when distributing programs that could be reverse-engineered. Indeed, this makes code understanding a challenging task and code replication hard to achieve to avoid code plagiarism and stealing specific functionalities; ② malware writers usually rely on obfuscation to harden manual reverse-engineering and make static analysis a more difficult task. Indeed, malware writers can rely on obfuscation to hide strings that would only be deciphered at execution time or to hide API calls with a reflection mechanism.

1.3 Contributions

In this chapter, we summarize the contributions of this dissertation as follows:

- **A replication study of a static logic bomb detector for Android apps.** We implement TSOpen, our open-source version of the unavailable TriggerScope static logic bomb detector. We perform a large scale study of TSOpen on more than 500 000 Android applications. Results indicate that the approach scales. Moreover, we investigate the discrepancies observed during our experiments using TSOpen and the original results and show that the approach can reach a very low false positive rate, 0.3%, but at a particular cost, e.g., removing 90% of sensitive methods considered during the control dependency step. Therefore, it might not be realistic to rely on such an approach to automatically detect *all* logic bombs in large datasets. However, it could be used to speed up the location of malicious code, for instance, while reverse engineering applications.

  Overall, TSOpen is a static analysis tool that is designed to identify logic bombs which are difficult to detect using traditional dynamic analysis techniques, as they are designed to evade detection by triggering only under specific circumstances. In addition, TSOpen improves the comprehensiveness of Android app analysis by exposing code that is hidden from existing dynamic analyzers.
  This work has led to the publication of a research paper in the IEEE Transactions on Dependable and Secure Computing journal in 2021 (TDSC'21).

- **An investigation of a hybrid approach to uncover suspicious hidden sensitive operations in Android apps.** We propose to investigate Suspicious Hidden Sensitive Operations (SHSOs) as a step towards triaging logic bombs. To that end, we develop a novel hybrid approach that combines static analysis and anomaly detection techniques to uncover SHSOs, which we predict as likely implementations of logic bombs. Concretely, Difuzer identifies SHSO entry points using an instrumentation engine and an inter-procedural data flow analysis. Then, it extracts trigger-specific features to characterize SHSOs and leverages One-Class SVM to implement an unsupervised learning model for detecting abnormal triggers. We evaluate our prototype and show that it yields a precision of 99.02% to detect SHSOs among which 29.7% are logic bombs. Difuzer outperforms the state-of-the-art in revealing more logic bombs while yielding less false positives in about one order of magnitude less time.

  Overall, Difuzer is a significant step forward in the challenging task of detecting malicious code triggered under specific circumstances in Android apps. Indeed, its ability to statically detect malicious code that is hidden from dynamic analyzers is a key factor in improving analyses' comprehensiveness and the overall security and reliability of Android apps.
  This work has led to the publication of a research paper in the 44^{th} IEEE/ACM International Conference on Software Engineering in 2022 (ICSE'22).

- **The creation of a new dataset of Android apps automatically infected with logic bombs.** We present TriggerZoo, a new dataset of 406 Android apps containing logic bombs and benign trigger-based behavior that we release only to the research community using authenticated API. These apps are real-world apps from Google Play that have been automatically infected by our tool AndroBomb. The injected pieces of code implementing the logic bombs cover a large pallet of realistic logic bomb types that we have manually characterized from a set of real logic bombs.

Researchers can exploit this dataset as ground truth to assess their approaches and provide comparisons against other tools.

Overall, TriggerZoo represents an important contribution to the research effort to detect malicious code in Android apps, as it provides a previously non-existent dataset that can be used to develop and evaluate new approaches to detecting logic bombs.
This work has led to the publication of a research paper in the 19th International Conference on Mining Software Repositories (MSR'22).

- **A new approach to account for atypical inter-component communication in Android apps.** We propose RAICC, a static approach for modeling new ICC links and thus boosting previous analysis tasks such as ICC vulnerability detection, privacy leaks detection, malware detection, etc. We have evaluated RAICC on 20 benchmark apps, demonstrating that it improves the precision and recall of uncovered leaks in state of the art tools. We have also performed a large empirical investigation showing that Atypical ICC methods are largely used in Android apps, although not necessarily for data transfer. We also show that RAICC increases the number of ICC links found by 61.6% on a dataset of real-world malicious apps, and that RAICC enables the detection of new ICC vulnerabilities.

  Overall, RAICC is designed to tackle the challenging task of providing better modeling of Android apps for static analysis. By enabling the detection of new ICC links, RAICC is able to reach program points that were previously not reachable during data flow analysis, leading to results that were previously missed by other approaches. This ability to more accurately model the complex interactions within Android apps is a key factor in improving the comprehensiveness and reliability of static analysis techniques.
  This work has led to the publication of a research paper in the 43rd IEEE/ACM International Conference on Software Engineering in 2021 (ICSE'21).

- **A step towards Android code unification in Android apps.** We propose a new advance in the ambitious research direction of building a unified model of all code in Android apps. The JuCify approach presented is a significant step towards such a model, where we extract and merge call graphs of native code and bytecode to make the final model readily-usable by a common Android analysis framework: in our implementation, JuCify builds on the Soot internal intermediate representation. We performed empirical investigations to highlight how, without the unified model, a significant amount of Java methods called from the native code are "unreachable" in apps' call graphs, both in goodware and malware. Using JuCify, we were able to enable static analyzers to reveal cases where malware relied on native code to hide invocation of payment library code or of other sensitive code in the Android framework. Additionally, JuCify's model enables state-of-the-art tools to achieve better precision and recall in detecting data leaks through native code. Finally, we show that by using JuCify we can find sensitive data leaks that pass through native code.

  Overall, JuCify represents a significant step forward in static analysis of Android apps. By tackling the challenges of providing a better static app model that can reason as closely as possible to runtime behavior, and accounting for the multi-language trend in apps, JuCify enables more comprehensive static analysis of Android apps. This is expected to have a significant impact on the field of Android app security, as it will allow for more accurate and reliable detection of vulnerabilities and malicious code.

This work has led to the publication of a research paper in the 44^{th} IEEE/ACM International Conference on Software Engineering in 2022 (ICSE'22).

- **A novel approach to account for conditional implicit calls in Android apps.** We investigated *conditional* implicit calls and their triggering criteria in the Android framework. We developed and evaluated ARCHER, a tool that resolves conditional implicit calls and extracts the constraints that trigger the delegation of execution control. Our empirical study shows that ① conditional implicit calls are widespread in Android applications; ② ARCHER allows to cover previously unreachable code compared to state-of-the-art approaches; and ③ ARCHER aids dynamic analyzers in covering conditional implicit calls by inferring the constraint values that lead to their triggering.

  Overall, ARCHER achieves multiple goals in the field of Android app security. By adding previously unknown edges in the call graph, it allows for more sound static analysis. Additionally, by resolving the targets of implicit calls and avoiding over-approximation and scaling issues, ARCHER enables more precise static analysis. Finally, by extracting appropriate inputs to reach implicit call targets, ARCHER improves the comprehensiveness of dynamic analyzers. By addressing these challenges, ARCHER makes a significant contribution to the field of Android app security, enabling more comprehensive and reliable analysis of these apps to detect vulnerabilities and malicious code.
  This work has led to a research paper that has been submitted to the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).

1.4 Roadmap

The roadmap of this dissertation is available in Figure 1.3. First, we introduce the necessary background information related to Android, static analysis and taint analysis in chapter 2. Then, we present part I that aims at describing how we contributed to expose code that is unreachable for dynamic analyzers. In particular, chapter 3 gives the motivation of part I and chapter 4 introduces the necessary background related to part I. In chapter 5 we present our replication study in which we implement our own version of an unavailable static logic bomb detector and highlight the discrepancies between our and the original results. Then, in chapter 6, we present our hybrid approach to detect suspicious hidden sensitive operations toards triaging logc bombs using code instrumentation, taint analysis and anomaly detection. Chapter 7 presents a dataset of Android apps automatically infected with logic bombs that we provide to the community. We conclude part I in chapter 8.

Subsequently, we dive into part II that aims at describing how we contributed to expose code that is unreachable for existing static analyzers. Chapter 9 motivates this part. In chapter 10, we present our work to account for atypical inter-component communication in Android apps. Then, chapter 11 presents our approach to account for native code for enhanced static analysis. We introduce our work on conditional implicit calls in chapter 12 in which we show how our approach improves both static and dynamic analyzers. Chapter 13 concludes part II on improving the comprehensiveness of Android apps' static analysis.

We broaden the horizons for future work in chapter 14. Eventually, we conclude this dissertation in chapter 15.

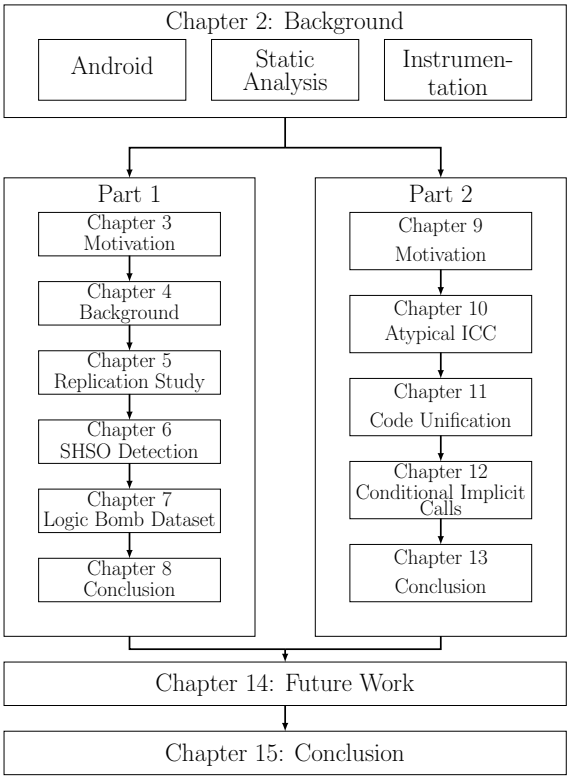


Figure 1.3: The roadmap proposed for this dissertation.

Chapter 2

Background

In this chapter, we introduce the concepts necessary for understanding the intention, the technical details, and the key concern of the research studies presented in this manuscript.

Contents

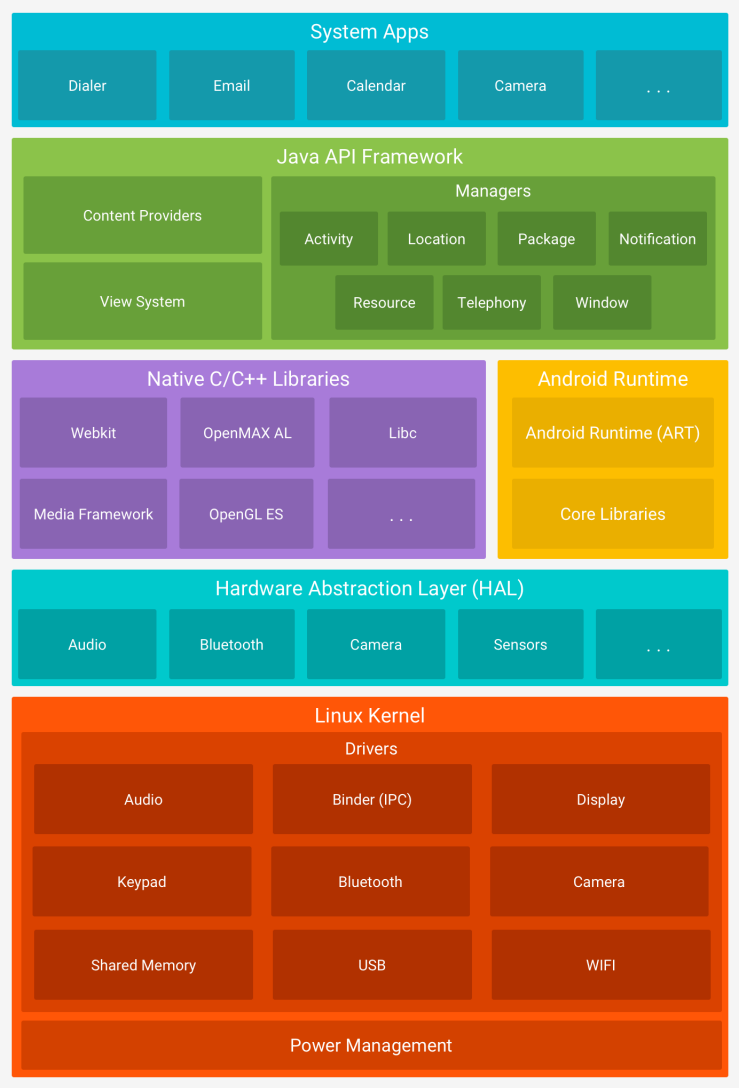


Figure 2.1: The Android Software Stack (from: developer.android.com/guide/platform)

2.1 Android

In this section, we briefly introduce background knowledge about Android. First, we give an overview of the Android operating system. Then, we present the Android framework and its interaction with Android apps. Eventually, we describe a key concept in Android apps, i.e., inter-component communication.

2.1.1 Operating System

Android is an open-source operating system built on top of the Linux kernel and made for mobile and commodity devices. It is composed of different components built on top of each other and forming a software stack that we can see in Figure 2.1.

The first layer is the Linux kernel on which Android is built to reuse a secure, portable, reusable, and free environment. Indeed, the Android runtime directly relies on this layer to perform low-level tasks such as accessing drivers, threading, IPC, etc. Also, the Linux kernel has been widely used for years and is used in millions of security-sensitive environments. Thus, Android takes advantage of the Linux kernel's security features.

Devices embed a profusion of elements to interact with users and provide interesting information, such as speakers, cameras, sensors, etc. To communicate with these elements,

File/Folder	Description
META-INF/	Metadata of the application, e.g., certificate files
lib/	Compiled code that is platform dependent
res/	Resources that are not compiled, e.g., pictures.
assets/	Application assets
AndroidManifest.xml	Main application configuration file
classes.dex	compiled Dalvik bytecode
resources.arsc	Compiled resources, e.g., binary XML files

Table 2.1: Main content found in Android Packages

the Android Software Stack provides a Hardware Abstraction Layer that features standard interfaces that can be used by the higher Java API framework layer. Hence, developers can easily use library modules to interact with hardware components.

Android apps run in their own process as well as their own instance of the Android runtime to ensure clear separation between apps. Indeed, the Android runtime is capable of running multiple virtual machines at once. Even though apps are mainly written in Java, they are not compiled in Java bytecode. Rather, the Java code is compiled into a Dalvik bytecode which is an optimized version of the Java bytecode designed for systems contrained in memory and CPU. The Dalvik bytecode is stored in DEX files that are found in Android apps and loaded by the Android runtime to execute the app.

Before the Java API framework, there is a layer of Native C and C++ libraries. These libraries are exposed to the Java API framework with custom API methods to allow developers to reuse functionalities available in the Native libraries. For instance. developers can access OpenGL through the Android framework to support the manipulation of 2D and 3D graphics. Besides, Android provides a Native Development Kit (NDK) to access native libraries and allows developers to develop their apps with native functions.

The Java API framework, i.e., the Android framework, is a set of API methods written in Java to ease app development to developers. The Android framework provides developers with convenient interfaces and classes to build their apps, e.g., `Activity` classes to provide UI to end users. In other words, through the reusability paradigm, these API methods are the building blocks to creating Android apps.

Lastly, the app layer represents both the system and developer apps. System apps come with the Android framework and act as a core of essential apps for a mobile device, e.g., messaging, browser, calendar, etc.

In this dissertation, our work mainly focuses on analyzing Android apps. More precisely, our static analysis techniques concentrate on the Dalvik bytecode and, in Chapter 11, on native code in Android apps.

2.1.2 App composition

Android apps are delivered to users as an *Android Package* format (APK). An APK file is no more than a collection of files archived as a zip file. Although many files can be found in such a package, there are several files and folders that are the backbones of Android apps. We list the main components that can be found in APK files in Table 2.1.

2.2 Static Analysis

Static analysis consists of analyzing a program without executing it. It is used to discover different semantic properties of programs. For instance, static analysis can be used to

answer a question such as: is this variable ever initialized in the program? Static analysis is central in this dissertation and our work in which we make thorough use. Indeed, we use static analysis to: ① cope with dynamic analysis limitations in terms of logic bomb detections; and ② expose code that is hidden for existing static analyzers to improve Android apps' comprehensiveness.

To perform our static analyses, we relied on existing tools and frameworks for modeling Android apps. To that end, we heavily leverage the Soot [24] static analysis framework that is able to load Dalvik bytecode and transform it into an intermediate representation, i.e., Jimple to construct control flow graphs on which intra-procedural static analysis techniques can be applied. In addition to that, we relied on the Flowdroid [5] static taint analyzer, itself built on top of Soot, which embeds an engine to properly model the control flow of Android apps due to life cycle and callback methods. Soot provides implementations of call graph construction algorithms such as CHA, RTA, VTA, or SPARK that are helpful in performing inter-procedural analyses. In this section, we give key elements about static analysis and the aforementioned toolchains used in our work.

2.2.1 Need for an Intermediate Representation of the Code

Analyzing Java source code is challenging for two reasons: ① it is not optimized for static analysis, i.e., its syntax is not *flat* and possesses more than 200 different opcodes that would need to be taken into account one by one in a data flow analysis to apply rules according to the semantic of the opcode; and ② it requires the availability of apps' source code, which is rarely the case.

Android apps are distributed in the form of APK files where the Dalvik bytecode is directly accessible. However, likewise Java, the Dalvik bytecode is complex to analyze since it has been made for runtime performance, not static analysis. Indeed, the Dalvik bytecode is register-based, which challenges static analyzers to propagate data and does not show any explicit type for variables.

To cope with these limitations, researchers created the *Jimple* (**J**ava s**imple**) intermediate language, which transforms the Dalvik bytecode into an intermediate representation optimized for static analysis. Indeed, Jimple has the following interesting properties:

- it contains only 15 different types of statements
- it is explicitly typed
- it is based on a three-address code
- it is flat (i.e., no nested code)

An example is available in Figures 2.2, 2.4 and 2.3 where we see the Fibonacci recursive version in three different representations. This example shows the advantages of using an intermediate representation for static analysis, i.e., the Jimple representation in this case.

The conversion from Dalvik bytecode to Jimple does not lead to semantic loss. In our work, we relied on the Jimple transformer implemented in Soot to load Android apps and work on their Jimple version.

2.2.2 Control Flow Graph

As usual in computer science, analysts work with abstract data types, such as graphs which allow them to represent and process data more easily. Static analysis is no exception to the rule. Indeed, even with a simplified language, i.e., Jimple, static analyzers do not work at the text level. The Jimple intermediate representation is loaded as a control flow graph representation which was introduced by Frances E. Allen in 1970 [25]. In these graphs, a node represents a statement (or an instruction), and an edge represents the control flow

```
public class Fib {

    public static void main(String args[]) {
        System.out.println(fib(9));
    }

    public static int fib(int n) {
        if (n <= 1){
            return n;
        }
        return fib(n - 1) + fib(n - 2);
    }
}
```

Figure 2.2: Fibonacci program in the Java source code representation

```
public class Fib extends java.lang.Object {
    public void <init>() {
        Fib r0;
        r0 := @this: Fib;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return;
    }

    public static void main(java.lang.String[]) {
        java.io.PrintStream $r0;
        int $i0;
        java.lang.String[] r1;
        r1 := @parameter0: java.lang.String[];
        $r0 = <java.lang.System: java.io.PrintStream out>;
        $i0 = staticinvoke <Fib: int fib(int)>(9);
        virtualinvoke $r0.<java.io.PrintStream: void println(int)>($i0);
        return;
    }

    public static int fib(int) {
        int i0, $i1, $i2, $i3, $i4, $i5;
        i0 := @parameter0: int;
        if i0 > 1 goto label1;
        return i0;
     label1:
        $i1 = i0 - 1;
        $i2 = staticinvoke <Fib: int fib(int)>($i1);
        $i3 = i0 - 2;
        $i4 = staticinvoke <Fib: int fib(int)>($i3);
        $i5 = $i2 + $i4;
        return $i5;
    }
}
```

Figure 2.3: Fibonacci program in the Jimple representation

```
1  public class Fib {
2
3    public Fib();
4      Code:
5        0: aload_0
6        1: invokespecial #1
7        4: return
8
9    public static void main(String[]);
10     Code:
11       0: getstatic     #2
12       System.out:Ljava/io/PrintStream;
```

(a) Part 1

```
1          3: bipush        9
2          5: invokestatic  #3
3          8: invokevirtual #4
4         11: return
5
6    public static int fib(int);
7
8      Code:
9        0: iload_0
10       1: iconst_1
11       2: if_icmpgt     7
12       5: iload_0
```

(b) Part 2

```
1          6: ireturn
2          7: iload_0
3          8: iconst_1
4          9: isub
5         10: invokestatic  #3
6         13: iload_0
7         14: iconst_2
8         15: isub
9         16: invokestatic  #3
10        19: iadd
11        20: ireturn
12   }
```

(c) Part 3

Figure 2.4: Fibonacci program in the bytecode representation

```
1  public class MainActivity extends Activity {
2    @Override
3    protected void onCreate(Bundle b) {
4      Animal a;
5      if(condition){
6        a = new Cat();
7      } else {
8        a = new Dog();
9      }
10     a.eat(); // <-- a could either be a cat or a dog at this program point
11   }
12 }
```

Listing 2.1: An example of how polymorphism affects call graph construction

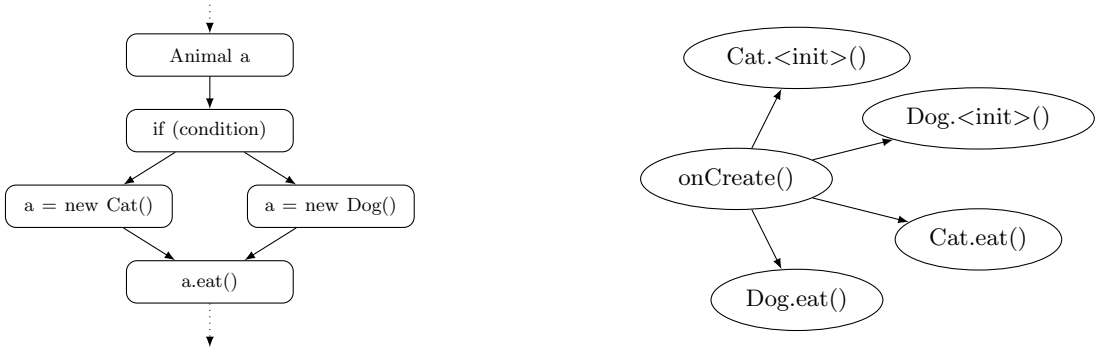
between two instructions. For instance, an edge e from node n_1 to node n_2 means that n_2 might be executed after node n_1. An example of a control flow graph is given in Figure 2.5a where it represents method `onCreate()` from Listing 2.1. Control flow graphs are easy to compute, with a node per statement/instruction and an edge per possible control flow in the program; With the Soot framework, control flow graphs represent the content of single methods to which analysts can apply intra-procedural analyses.

2.2.3 Call Graph

Intra-procedural analysis is not enough in static analysis. Indeed, analysts often perform inter-procedural analyses, i.e., analyses over the whole program, not methods independently. This implies that when a method is encountered during analysis, a jump is performed to the method called to be analyzed. To that end, call graphs representing the calling relationships between methods must be computed. In these graphs, a node represents a method, and an edge represents the relationship between two methods. For instance, an edge e from method m_1 to method m_2 means that m_2 might be called within method m_1. Contrary to control flow graphs, call graphs are not trivial to compute. Indeed, in object-oriented programs such as Java, call graphs are, in part, over-approximated since there are mechanisms that prevent computing a 100% precise call graph. For instance, with polymorphism, it is challenging to statically know the exact type of a particular object at a given statement. For instance, consider Listing 2.1 where a variable `a` is either a cat (line 6) or a dog (line 8). On line 10, `a`'s exact type cannot be statically inferred. Hence a could either be a cat or a dog. Hence, in the resulting call graph, the `onCreate()` method would be the source of both `Cat.eat()` and `Dog.eat()` as an over-approximation, see Figure 2.5b.

2.2.4 Taint Analysis

Taint analysis is a dataflow analysis that follows the flow of specific values within a program. A variable V is tainted when it gets a value from specific functions called *sources*. The taint is propagated to other variables if they receive a derivation of the value in V. If a tainted variable is used as a parameter of specific functions called *sinks*, it means that during execution, the value derived from a *source* can be used as a parameter of a *sink*. Taint analysis is often used to detect data leaks in Android apps. In the context of this dissertation, we often rely on the FLOWDROID's taint analysis engine to propagate and follow specific data in Android apps. For instance, in Chapter 6, we rely on taint analysis to check if pieces of data fall into if statements.



(a) Control Flow Graph

(b) Call Graph

Figure 2.5: Control Flow Graph and Call Graph examples of method `onCreate()` from Listing 2.1

2.3 Code Instrumentation

Code instrumentation refers to a technique used to add/remove/modify pieces of code of a program. For instance, one could statically modify the conditions of a program to force the execution of a given path. The instrumentation in the static Android apps analysis ecosystem is often used to modify apps and make them "more statically analyzable" since they heavily rely on implicit calls, which would hinder analyses [6, 26, 7, 27].

Part I

Statically Analyzing Code that is Difficult to Analyze for Dynamic Analyzers in Android Applications

Chapter 3

Motivation

Since the Android operating system is the most used in mobile devices and contains numerous sensitive data related to its owner, it is a target of choice for malware developers. The proof is that every year, thousands of new threats are identified by malware fighters (e.g., antivirus corporations and researchers) in Android apps in the form of, e.g., spyware, adware, ransomware, keylogger, etc. That is why, during the last decade, Android security and privacy have become important concerns in the research community that has proposed various techniques to fight against malware proliferation [12, 14, 28, 29, 30, 31]. As a consequence, malicious developers build their codebase to avoid detection from analyzers [32, 33, 17]. A notable technique used to bypass dynamic analyses consists in employing logic bombs that allow the malicious code to be triggered only under specific circumstances (e.g., at a specific date). In the following, we show how a malware developer can implement such malicious trigger behavior from three perspectives to highlight the importance of detecting logic bombs in Android apps.

3.1 Untargeted logic bomb

The first example in Listing 3.1 shows a piece of code that triggers the execution of malicious code after a specific data, i.e., a time-bomb. Hence, the malicious code would remain silent for some time before being triggered. Lines 6–8 depict how a time bomb can be implemented to bypass dynamic analyzers with its trigger condition on line 6 and its malicious guarded code in line 7. We can see that with a minimum of effort, a malware developer can bypass most of the dynamic analyses that study applications' behavior by monitoring them. This is an example of how malicious organizations can set up untargeted attacks in Android apps that would be widespread on Android devices, e.g., in games.

```
1  public class MainActivity extends Activity {
2    @Override
3    protected void onCreate(Bundle b) {
4      Date now = new Date();
5      Date attackDate = installDate.plusDays(14);
6      if(now.after(attackDate)){
7        // malicious code
8      }
9    }
10 }
```

Listing 3.1: Example of an untargeted logic bomb in an Android app

```
1  public class MainActivity extends Activity {
2    @Override
3    protected void onCreate(Bundle b) {
4      TelephonyManager tm = (TelephonyManager) this.getSystemService("phone");
5      String countryCode = tm.getNetworkCountryIso();
6      if(countryCode.equals("RU")) {
7        Camera.PictureCallback cb = new Camera.PictureCallback() {
8          public void onPictureTaken(final byte[] b, Camera c) {
9            // malicious code
10         }
11       }
12       if (Camera.getNumberOfCameras() >= 2) {
13         Camera.open(1).takePicture(null, null, cb);
14       }
15     }
16   }
17 }
```

Listing 3.2: Example of a coarse-grained targeted logic bomb in an Android app

```
1  public class MyBroadCastReceiver extends BroadcastReceiver {
2    @Override
3    public void onReceive(Context c, Intent i) {
4      SmsMessage sms = getIncomingSms(i);
5      String body = sms.getMessageBody();
6      if(body.startsWith("!CMD:")) {
7        // malicious code
8      }
9    }
10 }
```

Listing 3.3: Example of a fine-grained targeted logic bomb in an Android app

3.2 Coarse-grained targeted logic bomb

if attackers want to target a specific group of people, e.g., devices in Russia with two cameras (i.e., a back and a front), they can rely on the code illustrated in Listing 3.2. Indeed, on line 6, the developer verifies if the device is connected to a Russian mobile network (the ISO-3166-1 alpha-2 country code RU represents the Russian country). On lines 12–14, a photograph is taken from the front camera if the device has at least two cameras, and some piece of malicious code is triggered on line 9. In this case, the first specific circumstance is that the device is connected to a Russian mobile network (i.e., the Russian population is targeted), and the second one is that the device contains two cameras (i.e., eliminating emulators and old devices.). This example shows, again, that the malicious code is only executed on specific conditions that are unlikely to happen when dynamically analyzing the app in a sandbox.

3.3 Fine-grained targeted logic bomb

Let's now consider a State-Sponsored Attack or an Advanced Persistent Threat [34] which targets specific devices. Those devices could embed seemingly legitimate applications that contain, e.g., an SMS-bomb and remain undetected by most of the analysis tools. Such a logic bomb could be used as a backdoor to steal sensitive user data to very specific users, e.g., politicians. Indeed, if it is a well-prepared targeted attack, the attacker could send an SMS with a specific string recognized by the application. Then the application

would leak wanted information and stop the broadcast of the SMS to other applications supposed to receive it. Listing 2 shows an implementation of such a threat in the method `onReceive()` of `BroadcastReceiver` class, which is triggered when the device receives an SMS. First, the body of the SMS is retrieved on line 4. Then, line 4 compares it against "!CMD:" which is a hardcoded string matched against the body of an SMS. This check is considered suspicious because the application can match a hardcoded string against any incoming SMS. Hence it can wait for external commands to be executed by the malicious part of the application. Note that it could be harmless. If the condition is satisfied, the command is retrieved from the SMS, and the malicious code is activated on line 7.

These examples show how important it is to detect as many logic bombs as possible in Android applications since it is possible to control or exfiltrate many personal and sensitive data. However, they also show the weakness of dynamic analysis to identify them which highlights the importance of using static analysis for this task. In this part, our work consists of: ① replicating an existing and unavailable static approach to detect logic bombs in Android apps and stressing the discrepancies between the results published and observed; ② proposing a new hybrid approach based on static analysis and anomaly detection to identify suspicious hidden sensitive operations toward triaging logic bombs in Android apps; ③ building a new dataset of Android apps automatically infected with logic bombs.

Chapter 4

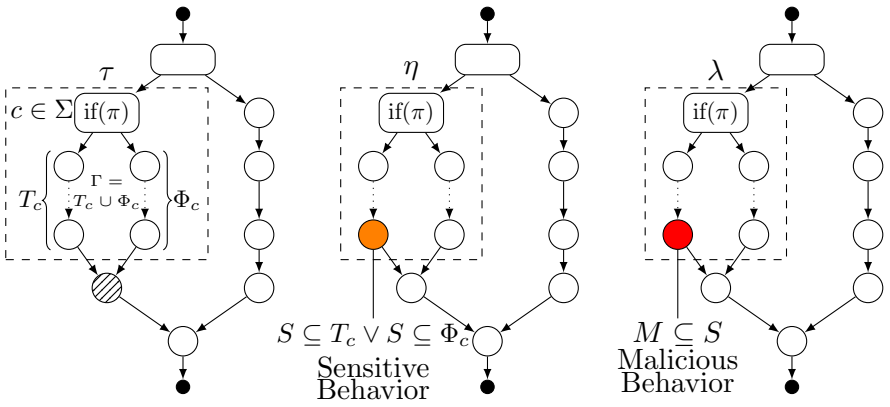
Background

In this chapter, we introduce the concepts that are necessary to understand our work on logic bomb detection.

4.1 Logic bomb

A logic bomb is a piece of malicious code triggered under very specific circumstances. It means that the malicious code is segregated from the normal execution of the benign code and is only triggered under specific criteria. Let us formally define what a logic bomb is.

Definitions: We define terms that will be used and referred to throughout the dissertation. Figure 4.1 visually depicts our definitions.



(a) Trigger *Definitions 1, 2, 3* **(b) Hidden Sensitive Operation** *Definitions 4, 5* **(c) Logic Bomb** *Definition 6*

Figure 4.1: Definitions illustrations. The graphs represent the Control Flow Graph of the same function.

Definition 1 (Trigger). A trigger is a piece of code that activates operations under certain conditions. In Figure 4.1a, the trigger τ (dashed rectangle) is represented by the condition c (rounded rectangle node), the true branch T_c and the false branch Φ_c. The true branch T_c represents all the statements (nodes) for which each path from the entry point must go through c and are executed if and only if π is true. Note that every path from the entry point to the hatched node must go through c. In other words, c strictly dominates the hatched node. However, the hatched node can be executed if π is true or false. Therefore it is not part of T_c nor Φ_c. The false branch Φ_c represents all the statements for which

each path from the entry point must go through c and are executed if and only if π is false.

More formally, let Σ be the set of statements of a function (nodes in Fig. 4.1). Let $c \in \Sigma$ be a conditional statement (i.e., an if statement, rectangle nodes in Fig. 4.1). Let π be c's predicate. Let ε be the conditional execution function such as $\varepsilon(\pi, \sigma)$ is true if $\sigma \in \Sigma$ is executed if and only if π is true. Let δ be the dominator function such as $\delta(d, \sigma)$ is true if $d \in \Sigma$ strictly dominates $\sigma \in \Sigma$, false otherwise.
Let T_c and Φ_c be the *true* and the *false* branch [1] of c such as:

$$T_c = \{\sigma \mid \sigma \in \Sigma \wedge \delta(c, \sigma) \wedge \varepsilon(\pi, \sigma)\}$$
$$\Phi_c = \{\sigma \mid \sigma \in \Sigma \wedge \delta(c, \sigma) \wedge \varepsilon(\neg\pi, \sigma)\}$$

Then, a trigger τ is defined as a triplet: $\tau = (c, T_c, \Phi_c)$.
Definition 2 (Guarded code). Let τ be a trigger such as: $\tau = (c, T_c, \Phi_c)$. Then, the code guarded by c is: $\Gamma = T_c \cup \Phi_c$.
Definition 3 (Trigger entry point). We define a trigger entry point as the condition triggering the guarded code. More formally, given a trigger $\tau = (c, T_c, \Phi_c)$, c is defined as its entry point.
Definition 4 (Hidden Sensitive Operation (HSO)). An HSO is a piece of code that represents a set of instructions, which (1) implement a security-sensitive operation and (2) are only executed when specific criteria are met (cf. Figure 4.1b). More formally, let $\eta = (c, T_c, \Phi_c)$ be a trigger and S a piece of sensitive behavior such as $S \subset \Sigma$. Then, η is a hidden sensitive operation if $S \subseteq T_c \vee S \subseteq \Phi_c$.
Definition 5 (Suspicious Hidden Sensitive Operation (SHSO)). An SHSO refers to an HSO that implements a sensitive operation that appears to be suspicious given the context of the app. For example, a navigation app may legitimately retrieve user location information (which is a sensitive operation), while a calculator is suspicious if it attempts to retrieve such sensitive data.
Definition 6 (Logic bomb). A logic bomb is a piece of malicious code triggered under specific circumstances. More formally, let $\lambda = (c, T_c, \Phi_c)$ be an SHSO, S its sensitive behavior, and M a piece of malicious code such as $M \subset \Sigma$. Then, λ is a logic bomb if $M \subseteq S$ (cf. Figure 4.1c). In other words, a logic bomb is an SHSO which suspicious sensitive behaviour is malicious.

4.2 Anomaly detection

When analyzing data of the same class, several items can significantly differ from the majority. They are called *outliers* and can be viewed as abnormal. There are numerous techniques in the state-of-the-art for achieving this outlier detection in sets of data [35]. Our work relies on *One-Class Support Vector Machine* (OCSVM) [36], an unsupervised learning algorithm that learns common behavior based on features extracted in an initial dataset. Once the model is learned, a prediction is performed by checking whether a new sample features make it more or less abnormal w.r.t. the common model. In this dissertation's context, an anomaly is computed by considering distances among vectors representing *triggers*, i.e., a condition along with the behavior triggered.

4.3 Symbolic Execution

The notion of symbolic execution used in this part is taken from the TriggerScope's paper [17] since our first work presented is a replication of TriggerScope. In this work,

[1] Note that in case there is no false branch, $\Phi_c = \emptyset$.

the static analysis engine has to make decisions based on the condition's semantics, i.e., what the values used in the condition possibly refer to. The analysis models the values and the operations performed on objects by tagging them. For instance an object receiving the results of the call to `Date.getHour()` would be annotated with `#now/#hour`.

Chapter 5

Replication of a Static Logic Bomb Detector for Android Apps

In this chapter, we propose to investigate a replication study to assess to what extent the approach presented can be relied upon to detect logic bombs in Android apps. To that end, we implemented TSOpen*, our open-source version of the unavailable* TriggerScope *static logic bomb detector. We investigate the discrepancies observed during our experiments using* TSOpen *and the results presented in the original paper. Results indicate that though the approach scales, a high false positive rate makes manual vetting challenging to find logic bombs in Android apps.*

This chapter is based on our work published in the following research paper:

- **Jordan Samhi**, Alexandre Bartel. On The (In)Effectiveness of Static Logic Bomb Detector for Android Apps. *IEEE Transactions on Dependable and Secure Computing (**TDSC**)*, 2021, 10.1109/TDSC.2021.3108057 [37].

Contents

5.1 Overview

Android is the most popular mobile operating system, with more than 71% of the market share in 2022 [38], which undeniably makes it a target of choice for attackers. Fortunately, Google set up different solutions to secure access for applications in their *Google Play*. It ranges from fully-automated programs using state-of-the-art technologies (e.g., Google Play Protect [39]) to manual reviews of randomly selected applications. The predominant opinion is that the *Google Play* market is considered relatively malware-free. However, as automated techniques are not entirely reliable and manually reviewing every submitted application is not possible, they continuously improve their solutions' precision and continue to analyze already present-in-store applications.

Consequently, attackers' main challenge is building malicious applications that remain under the radar of automated techniques. For this purpose, they can obfuscate the code to make the analysis more difficult. Example of obfuscation includes code manipulation techniques [40], use of dynamic code loading [41] or use of the Java reflection API [42]. Attackers can also use other techniques such as packing [43] which relies on encryption to hide their malicious code. This work focuses on one type of evasion technique based on logic bombs. A logic bomb is code logic that executes malicious code only when particular conditions are met.

A classic example would be malicious code triggered only if the application is not running in a sandboxed environment or after a hard-coded date, making it invisible for dynamic analyses. This behavior shows how simple code logic can defeat most dynamic analyses leading to undetected malicious applications.

In the last decade, researchers have developed multiple tools to help detecting logic bombs [30, 14, 44]. Most of them are either not fully automated, not generic or have a low recall. However, one approach, TriggerScope [17], stands out because it is fully automated and claims a false positive rate close to 0.3%. In this work, we reimplement TriggerScope and perform a large-scale study to replicate TriggerScope's results. Furthermore, we identify specific parameters that directly impact the false positive rate.

As TriggerScope is not publicly available and the authors cannot share the tool, we implement their approach as an open-source version called TSOpen. Although TSOpen has been implemented by faithfully following the details of the approach given in TriggerScope paper, we did not use the same programming language, i.e., C++. We used Java to reuse publicly available and well-tested state-of-the-art solutions. Indeed, our solution relies on the so-called Soot framework [24] to convert the Java bytecode into an intermediate representation called Jimple [45] and to automatically construct control flow graphs. Also, to model the Android framework, the life-cycle of each component, and the inter-component communication, TSOpen relies on algorithms from FlowDroid [5].

We use TSOpen to conduct a large-scale analysis to see if such a static approach is scalable. We ran TSOpen over a set of 508 122 applications from a well-known database of Android applications named Androzoo [46]. This experiment shows that the approach is scalable but yields a high false positive rate. Hence, because of this high false positive rate, the approach might not be suitable to detect *all* logic bombs automatically. More than 99 651 applications were flagged with 522 300 triggers supposedly malicious, yielding a false positive rate of more than 17%.

Since we obtained a false positive rate much higher than in the literature, we investigated the discrepancies. We construct multiple datasets to consider the concept drift effect, which could affect the results as shown by Jordaney et al. [47]. Moreover, we also investigate multiple aspects of the implementation, such as the list of sensitive methods, the call graph construction algorithm or the timeout threshold. Furthermore, we applied two additional filters not mentioned in the literature: (1) Purely symbolic values removal and (2) Different package name removal. Results indicate that to get close to a false posi-

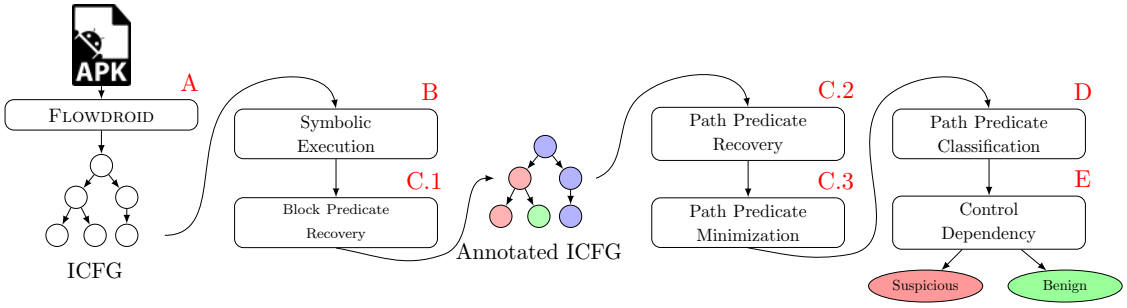


Figure 5.1: Overview TSOpen. First, the application APK is processed by FLOWDROID to model the application. Then, every step of the analysis is applied to the ICFG until the final decision of the application's suspiciousness is taken by our tool.

tive rate of 0.3%, either aggressive filters should be put in place or a short list of sensitive methods should be used. In both cases, the impact on the false negative rate is considerable. This means that if the approach is usable in practice with a low false positive rate, it might miss many applications containing logic bombs.

We have shared our work with the TRIGGERSCOPE's authors, who gave us positive feedback and did not see any significant issue regarding the approach or TSOPEN's design.

In summary, we present the following contributions :

- We implement TSOPEN, the first open-source version of the state-of-the-art approach for detecting logic bombs, and show that this approach might not be appropriate for **automatically** detecting logic bombs because it yields too many false positives.
- We conduct a large-scale analysis over a set of more than 500 000 Android applications. While the approach is theoretically not scalable because it relies on NP-hard algorithms, we find that, in practice, 80% of the applications can be analyzed.
- We conduct multiple experiments on the approach's parameters to see the impact on the false positive rate and identify that a low false positive rate can be reached but at the expense, for instance, of missing a large number of sensitive methods.
- We experimentally show that TRIGGERSCOPE's approach might not be usable in a realistic setting to detect logic bombs with the information given in the original paper. We empirically show that using TRIGGERSCOPE's approach, *trigger analysis* is insufficient to detect logic bombs.

We make available our implementation of TSOPEN with datasets to reproduce our experimental results:

`https://github.com/JordanSamhi/TSOpen`

5.2 Approach

In this section, we explain TSOPEN, an open-source implementation of TRIGGERSCOPE, at the conceptual level. The approach is summarized in Figure 5.1.

5.2.1 Applications representation

Android apps do not have a single entry point like usual Java programs. They are made of components, each with a life cycle managed by the Android framework.

Modeling life cycles and how components are connected is not trivial. That is why TSOpen relies on Flowdroid [5]. Indeed, Flowdroid handles intra-component communications by introducing dummy main methods and opaque predicates to guarantee that any execution order would not influence any static analysis over the model. Using state-of-the-art solutions allowed us to avoid re-implementing the Android framework modeling reducing implementation errors. Thus, we retrieve an *interprocedural control flow graph* on which we can run static analysis algorithms (step A in Figure 5.1).

Now that we have an application model (see Figure 5.2), we can run our analysis, starting with the symbolic execution.

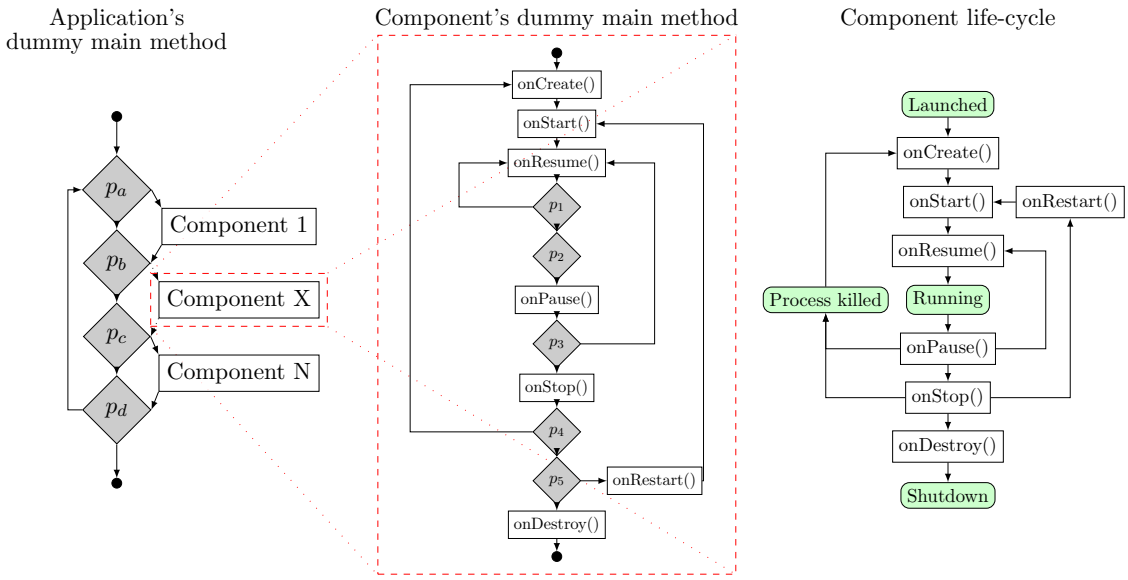


Figure 5.2: The left diagram represents the dummy main method of the entire application constructed by Flowdroid with each opaque predicate p_a, ..., p_d in gray. Opaque predicates will not be evaluated during analysis, hence both branches would be considered equally. Each component's life cycle is modeled after the corresponding life cycle. For instance, if Component X is an Activity component, FlowDroid models it according to the Activity life-cycle presented on the right-hand side of the figure. As for the dummy main, FlowDroid's concrete implementation of the component life-cycle (middle) contains opaque predicates.

5.2.2 Symbolic execution

When classifying predicates, the program has to make decisions depending on the type of objects in conditions, i.e., the condition's semantics. Therefore, this analysis models the values and operations performed over Java objects using symbolic execution (step B in Figure 5.1). More precisely, as we faithfully implemented TriggerScope, we focus on modeling strings, integers, location, SMS, and time-related objects. Also, these interesting objects are annotated to ease the classification.

Furthermore, the classification cannot be done without retrieving the instructions guarded by a condition, which is why the next step, i.e., the predicate recovery, is essential.

5.2.3 Predicate recovery

An essential step of the analysis is to construct the intra-procedural path predicate related to each instruction to build the logical formula leading to the instruction. For this, we

operate as follows:

Let $ICFG = (I_r, E_r)$ the directed graph describing the interprocedural control flow graph given by FLOWDROID where, I_r represents the set of reachable instructions of the program, and $E_r \subseteq I_r \times I_r$ corresponds to the set of reachable directed edges of the program represented by a pair of instructions (i_a, i_b) indicating that the flow goes from i_a to i_b. Let C_r the set of reachable conditions of the program and $\Gamma^-(i) = \{x \mid (x, i) \in E_r\}$ the predecessor function.

The algorithm to retrieve the full path predicate of each instruction is described as follows:

1. $\forall\, i \in I_r, \forall\, e = \{(x, i) \mid x \in \Gamma^-(i)\} \in E_r$, annotate e with the closest preceding condition $c \in C_r$ (step C.1 in Figure 5.1).

2. $\forall\, i \in I_r$ annotate i with $p = getFormula(i) =$ $\{\bigvee(getFormula(x) \wedge c) \mid x \in \Gamma^-(i)$, c the condition annotated on edge $(x, i)\}$ (step C.2 in Figure 5.1).

3. $\forall\, i \in I_r$, simplify the formula p with the basic laws of Boolean algebra, p is the full intraprocedural path predicate annotated on i (step C.3 in Figure 5.1).

The last step is essential to remove false dependencies of instructions. Indeed, consider the following formula: $(p \wedge q) \vee (\neg p \wedge q)$ which could have been calculated after step 2. The instruction annotated with this formula would have a false dependencies on predicate p because $(p \wedge q) \vee (\neg p \wedge q) = q \wedge (p \vee \neg p) = q \wedge 1 = q$ as defined by the distributive and complementation laws of boolean algebra. Hence, the elimination of false dependencies.

Now that we have retrieved path predicates and eliminated false dependencies, we can classify predicates.

5.2.4 Predicate classification

To classify predicates, i.e., their potential suspiciousness, two essential characteristics are taken into account (step D in Figure 5.1). Firstly, we verify that the predicate involves a previously computed time-, SMS- or location-related object. Secondly, we verify the type of check performed over the object. The focus is set to comparisons with relevant previously modeled objects and hardcoded values/constants in the application. If a condition corresponds to these criteria, it is flagged as suspicious. The Jimple intermediate representation of the Java bytecode is convenient for this stage as it allows analysts to access explicit object types. This step acts as a filter for the final control dependency step as it reduces the conditions to analyze by ruling out not suspicious conditions.

We can now perform the last step to check if a sensitive method is called within the guarded instructions of a suspicious condition.

5.2.5 Control dependency

The last step of the approach consists in characterizing whether a condition is defined as a logic bomb (step E in Figure 5.1). For this, every guarded instruction of a considered condition is checked to verify if it invokes a sensitive method. Also, TRIGGERSCOPE's developers had the idea to check whether a variable would be modified and later involved in another check, which, in turn, its guarded instructions would be similarly checked. This idea extends the range of possibilities regarding the search for logic bombs. Our implementation takes all these steps into account.

More information about the implementation is given in the following section.

5.2.6 Implementation

Any implementation is subject to erroneous code. We rely on well-tested art publicly available Java frameworks to reduce the number of errors in TSOpen. TriggerScope, on the other hand, is written from scratch in C++, which increases the risks of introducing numerous implementation errors [48].

TriggerScope has been initially developed in C++ with complete management of the transformation of the Dalvik bytecode in a custom intermediate representation on which the control flow analysis and the analysis are performed. The modeling of Android applications has well been explored in the state of the art, which is why we did not re-implement those aspects.

TSOpen consists of more than 5K Java SLOC (18.6K for TriggerScope) which, provides better results in term of execution time than the C++ version of TriggerScope. This is probably due to the choice, on their side, to implement the control flow analysis from scratch. Also, we parallelized, using multi-threading when it was possible, e.g., the symbolic execution and the path predicate recovery.

Besides using Flowdroid for modeling Android applications, our implementation is built on top of Soot [24] which is the state-of-the-art solution regarding static analysis over Java and Android [49] programs, initially described in 1999 and since used by researchers around the world.

Note that we do not possess the information about the call graph construction algorithm used by the authors of TriggerScope. Therefore, even though we try to faithfully implement the tool with the details of the paper, we cannot have a 100% similar tool. TSOpen relies on Flowdroid to construct the call graph used for the inter-procedural analysis. Flowdroid, in turn, relies by default on the Spark call graph analysis framework [50]. We evaluate TSOpen using different call graph construction algorithms as discussed in section 5.3.

Symbolic execution

Similarly to TriggerScope, our symbolic execution engine models numeric, string, time, SMS, and location-related objects. Again, the Jimple intermediate representation is convenient for recognizing objects and operations performed over those objects thanks to its flat representation of operations and its explicit typed variables. Furthermore, as our approach is built over Soot which allows optimizing the analyzed code, numeric and string constants are easily propagated, which facilitates the predicate classification.

Modeling string objects is important for detecting suspicious checks against any string value/field of an object, e.g., the body of an incoming SMS. For this purpose, our analysis models as faithfully as possible string values propagated along the graph. When dealing directly with concrete values, the analysis recognizes the operations performed and executes it directly, e.g., `append(), format(), subString()`, otherwise if it is a symbolic value it records the operation.

Regarding time-related objects, TSOpen keeps track of a list of time-related classes (e.g., `Calendar, Date, LocalDateTime, SimpleDateFormat, etc.`) and annotates each one with the tag `#now` when it recognizes that it has been instantiated with the current date in the program. Also, TSOpen keeps track of related methods that narrow the circumstances of the potential check performed on such values like `Date.getHour()` would be annotated with `#now/#hour` which eases the future classification.

Equivalently, TSOpen records every location-related objects like `android.location.Location` and annotates it with the `#here` tag when it is instantiated to represent the current location. Fields of those objects can be accessed to represent more precise values like the longitude or the latitude. TSOpen annotates those values with respectively

`#here/#longitude` and `#here/#latitude`.

Furthermore, the analysis makes the same approach for the `SmsMessage` object, i.e., it annotates it with the `#sms` tag and records, as strings, values retrieved from the received SMS. In this case, it keeps track of the use of the `getMessageBody(), getDisplayMessage-Body(), getOriginatingAddress()` and `getDisplayOriginatingAddress()` methods. Then it annotates them with `#sms/#body` or `#sms/#sender`.

TSOpen also models boolean values to keep track of methods returning a boolean value used in conditions.

For the context of this analysis, TSOpen annotates methods like `Date.after(), Date.before(), String.contains(), String.startsWith()`, etc.

Predicate recovery

This step aims to eliminate false dependencies while retrieving instructions dominated by the trigger condition which will be used for the control dependency. For this purpose, the analysis begins with a forward intra-procedural analysis, extracts simple predicates from each check, and annotates the edge corresponding with the predicate. An edge annotated with a predicate p means that the target of the edge must be executed if and only if p is satisfied.

The next step is the real predicate recovery; each node retrieves the list of predicates to be satisfied to reach this node. To this end, TSOpen performs a backward intra-procedural analysis and recursively combines the previous simple predicates. More precisely, a boolean formula is built following two rules: 1) if the node in question has one predecessor, it is combined using a logical AND; 2) if the node in question has more than one predecessor, it is combined using a logical OR between every possible path predicate.

Finally, the last step, without which the analysis would lead to a significant increase of false positives due to false dependencies, is the minimization of boolean formulas.

Predicate classification

Hitherto TSOpen has modeled interesting objects related to the purpose of this analysis and has propagated their values. It has also removed false dependencies. It now needs to decide whether a check is considered suspicious.

Time-related objects TSOpen verifies that (1) one of the operands is the result of the invocation of a comparison between two date/time objects such as `after()` or `before()` and (2) One represents the current date, and the other is built with a constant value. Similarly, it verifies that, in the check, some primitive numeric types representing the current date/time are compared with a constant. In these cases, it flags the check as suspicious.

Location-related objects Our tool verifies if one of the operands is a value derived from an object related to the current location and if the other operand represents a constant value. Also, it checks if one of the operands is the result of the invocation of a method such as `distanceBetween()` to check if the device is in a specific area. In these cases, it also flags the check as suspicious.

SMS-related objects TSOpen verifies if one of the operands represents a value from the body of an SMS or the sender of an incoming SMS. Then, if it is the case, it verifies if these values, which are strings, are matched against specific patterns or constants through the invocation of methods such as `startsWith(), endsWith(), contains(),`

`matches()`, etc. These checks are also flagged as suspicious because they encode tight conditions which could be used to exfiltrate data surreptitiously.

Furthermore, to set aside obvious non-suspicious checks like null check reference or the comparison of a number with "-1" (e.g., is the size of the body of an SMS greater than -1 ?), we apply a post-filter step as described in TriggerScope paper.

Control dependency

Finally, we have a reduced list of suspicious checks. We also have the list of instructions that are guarded by each check. The analysis can now determine if any of these checks contain an invocation to a sensitive Android API.

For this purpose, TSOpen iterates over guarded instructions of previously flagged suspicious trigger conditions and checks if they contain an invocation to a method. If it is the case, it verifies if this method appears in the list of sensitive methods considered in the paper of TriggerScope. The list is not public and not shared, but the researchers wrote that they used the result of PScout [51] and SuSi [52]. We also retrieved these lists and used them to classify a method as sensitive. The check is flagged as a potential logic bomb if a match is found.

Additionally, this approach is inter-procedural, meaning that the analysis will propagate to analyze the content of any method invocation to check if, in the call stack, a match can be found. Also, some malware do not invoke a sensitive method directly. Indeed, the logic bomb could be used to turn a switch (e.g., a boolean), and a check could be performed on this switch elsewhere in the code. That is why the analysis follows these updated fields between methods and checks whether they are guarded by a check which would invoke a sensitive method.

Execution example

```
1  public void onReceive(Context context,
   ↪ Intent i) {
2      SmsMessage sms = getSms(i);
3      //#sms
4      String b = sms.getBody();
5      //#sms/#body
6      String cmd = null;
7      //symbolic null value
8      if(b.startsWith("!CMD:")){
9          //#sms/#body.startsWith("!CMD:")
10         cmd = getCmdFromBody(b);
11         //symbolic string
12         processCmd(cmd);
13     }else{ // do something else
14     }
15 }
```

(a) Symbolic execution

```
1  public void onReceive(Context
   ↪ context, Intent i) {
2      SmsMessage sms = getSms(i);
3          // -
4      String b = sms.getBody();
5          // -
6      String cmd = null;
7          // -
8      if(b.startsWith("!CMD:")){
9          cmd = getCmdFromBody(b);
10         // p
11         processCmd(cmd);
12     }else{
13         // ¬p
14     }
15 }
```

(b) Path predicate recovery

```
1  public void onReceive(Context context,
   ↪ Intent i) {
2      SmsMessage sms = getSms(i);
3      String b = sms.getBody();
4      String cmd = null;
5      if(b.startsWith("!CMD:")){
6          // suspicious predicate
7          // code guarded by p
8          cmd = getCmdFromBody(b);
9          processCmd(cmd);
10         // sensitive method in
11         // inter-procedural call
12     }else{
13         // code guarded by ¬p
14     }
15 }
```

(c) Control dependency

Figure 5.3: Example of the different steps of the analysis

We explain the process of the approach with Figure 5.3. First, we describe the symbolic execution step with the example of Listing 5.3a. The first value modeled is in line 2. A new incoming SMS is being stored in variable *sms* and is represented by the tag `#sms`. In line 4, the body of the SMS is retrieved. Thus the instruction is tagged with `#sms/#body` as our symbolic execution engine recognizes it. Some values cannot be resolved during this step, hence they are assigned symbolic values, e.g. lines 6 and 10. Those modeled values are useful to describe the semantic of the condition at line 8 which is represented by the tag `#sms/#body.startsWith("!CMD:")`. This value will be used during the path predicate classification to qualify the suspiciousness of the condition.

Now that the analysis has tagged some interesting values, it retrieves the path predicate for each instruction in the code, as illustrated in Listing 5.3b. This simple example

shows that outside any conditions, intra-procedural instructions are not annotated with any logical formula (lines 3, 5, and 7). However, instructions guarded by a condition are annotated with the predicate representing this condition. Both branches are considered, which is why the instruction at line 9 is annotated with predicate p, and any instruction under the *else* instruction is annotated with $\neg p$. Those formulas are then subjected to minimization in order to rule out false dependencies. It is now possible to classify predicates and to check if they guard sensitive operations.

The next phase, shown in Listing 5.3c, allows classifying predicates thanks to the results of the previous symbolic execution. Indeed, decisions about the suspiciousness are taken according to the results of the symbolic execution. That is why the condition at line 5 is considered suspicious because the body of an incoming SMS is being matched against a hardcoded string. Once the suspicious conditions are memorized, the analysis retrieves the instructions guarded by those conditions thanks to the path predicate recovery step. For each guarded instruction, the analysis checks if the instruction is a method call and, if it is the case, the called method is being matched against a list of sensitive methods. If no match is found, as it is the case in 5.3c, the inter-procedural mechanism takes place, meaning the analysis dives into application method calls to check if they use a sensitive method. In this example, the `processCmd(String)` method at line 9 contains such a method call. According to TriggerScope's approach, it is sufficient to qualify this sequence as a logic bomb.

5.3 Evaluation

In this section, we evaluate TSOpen and address the following research questions:

RQ1: Does TSOpen's approach scale?

RQ2: What parameters can impact the false positive rate?

RQ3: Is it possible to locate the malicious code with logic bomb detection?

RQ4: Do benign and malicious applications use similar behavior regarding the approach under study and why?

RQ5: Are TriggerScope's results reproducible?

Our analyses were run on a server with an Intel Xeon E5-2430 2.20GHz processor with 24 cores, and 95GB of RAM and the *High-Performance Computing* [53] equipment available at the University of Luxembourg.

5.3.1 TSOpen scalability

We perform the large-scale analysis on a large dataset containing 508 122 applications. This dataset has been created by randomly selecting applications from the 10 million applications of Androzoo [46]. This analysis is necessary to understand why it could or could not be deployed in real-world analyses, e.g., before an app is accepted into an app market.

Analyzing millions of applications with a 1-hour timeout has to be parallelized to take as short a time as possible. For this purpose, we took advantage of the *High-Performance Computing* [53] equipment available at the University of Luxembourg.

We took into consideration 508 122 benign and malicious applications. Out of 508 122 considered applications, 405 810 (79.9%) were successfully analyzed with an average of 21 seconds per analysis. The proportion of applications with detected triggers with this

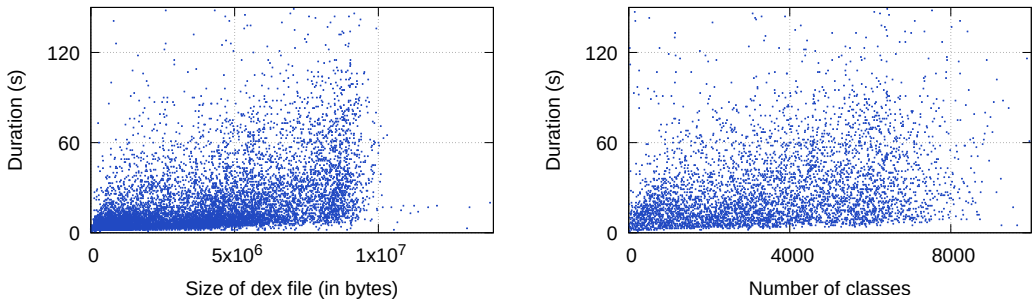


Figure 5.4: Evolution of the duration of the analysis depending on the size of the dex file and the number of classes in the applications considered.

approach is 19.6% (99 651) and 34.9% (177 112) without library filter (The library filter is further explained in Section 5.3.2). Also, 522 300 (791 364 without library filter) triggers are detected by TSOpen among which 0.48% of SMS-related triggers, 1.35% location-related triggers, and 98.17% of time-related triggers.

Our default threshold for the timeout is one hour. Some applications cannot be analyzed within one hour. A malware developer could simply use techniques, such as obfuscation, to slow down static analysis tools to prevent the application from being analyzed. To understand how an attacker could bypass the analysis, we measured the execution time of the analysis in function of four features: (a) the size of dex files, (b) the number of classes, (c) the number of objects, and (d) the number of branches.

In Figure 5.4, we can intuitively assume that there is no correlation between the size of the dex file or the number of classes in the app with the duration of the analysis. In fact, when measuring the *Pearson Correlation Coefficient (PCC)*, computed based on Equation 5.1, we can state that there is no correlation.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{5.1}$$

In Equation 5.1, n is the number of data pair, x_i and y_i are data points, $\bar{x}$ and $\bar{y}$ respectively correspond to $\frac{1}{n}\sum_{i=1}^{n} x_i$ and $\frac{1}{n}\sum_{i=1}^{n} y_i$.

The correlation coefficient computed for the data corresponding to the duration as a function of the dex size is equal to 0.152.

With its value close to 0, we can say that the size of the bytecode of an application does not influence the duration of the analysis, this means that even if an attacker naively introduces libraries or code to bring noise in the analysis, e.g., with dead code, it will not force the analysis to reach the timeout. Similarly, this type of analysis does not seem to be sensitive about the number of classes in an application as the *PCC* computed for the data corresponding to the duration as a function of the number of classes is equals to 0.152. The same conclusion can be done as for the dex file size, even with a lot of noise, meaning many classes brought by obfuscation, for example, the analysis still stays efficient.

On the other hand, other features directly influence the duration of the analysis. In Table 5.1 representing the correlation coefficients of Figure 5.5 we can see that the more objects in an application, the more time it will take to analyze the application. Indeed, while the *Pearson correlation coefficient* does not indicate any linear correlation (we can intuitively see the exponential correlation in Figure 5.5) due to the *PCC* value of 0.359, the *Spearman Correlation Coefficient* computed based on Equation 5.2 assures us that the relationship between the variables observed can be represented using a monotonic function [54] due to a coefficient of 0.908.

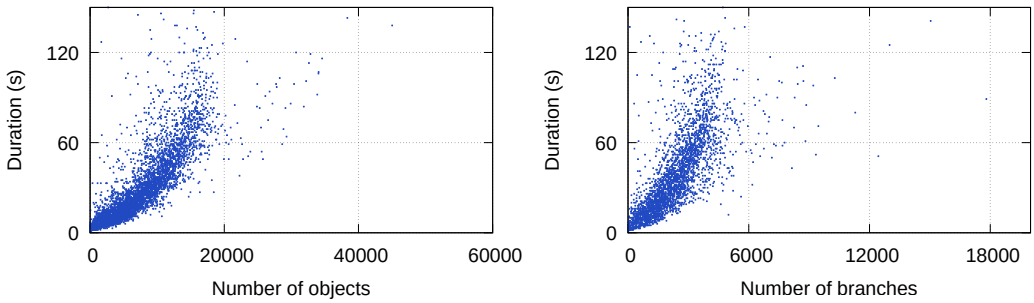


Figure 5.5: Evolution of the duration of the analysis depending on the number of objects and branches in the applications considered.

	PCC	**SCC**	**PCC** $(x_i, log(y_i))$
# of objects to time	0.359	0.908	0.795
# of branches to time	0.331	0.839	0.657

Table 5.1: Correlation coefficients of the data of Figure 5.5 (PCC: Pearson Correlation Coefficient, SCC: Spearman Correlation Coefficient)

$$r_s = \frac{cov(rg_x, rg_y)}{\sigma_{rg_x}, \sigma_{rg_y}} \tag{5.2}$$

In Equation 5.2 rg_x and rg_y respectively represent the rank variables of x and y. Similarly, σ_{rg_x} and σ_{rg_y} respectively represent the standard deviations of rg_x and rg_y.

Better, as exponential functions can be approximated into linear functions by taking the logarithm of both sides, we can compute a linear correlation coefficient on $(x_i, log(y_i))$ for $i \in \{0, 1, ..., n\}$ (n being the number of pairs of data) and extrapolate the results for the original data. We obtain a score of 0.795, which is a strong linear correlation, assuring us that the original data is positively correlated following an exponential function.

The explanation for this exponential correlation is simple: to understand and detect logic bombs, this approach aims at retrieving the semantic of objects of interest. This means that the more objects to model, the more statements to take into account while modeling, therefore the more time the analysis will take. This can be problematic for an application with many objects or an application where the developer deliberately introduces useless objects to introduce noise in the analysis.

Additionally, we can see in Table 5.1 that the number of reachable branches and the duration of the analysis are, similarly to the number of objects and the time of the analysis, positively correlated following an exponential function. Indeed, it introduces new paths, meaning many values to remember depending on the path during the symbolic execution. Regarding the approach used in this analysis, the most troublesome consequence of having many branches (path minimization is an NP-hard problem) is that it considerably slows down the analysis.

Furthermore, as the path predicate recovery is not necessary over the entire code of an application, we collected, afterward, on a subset of the large-scale study's applications the average time taken by the predicate recovery step. It revealed that it is responsible for 22.2% of the analysis time of an application on average and also responsible, in 35.3% of the cases, for reaching the timeout. In contrast, the symbolic execution is responsible for 61.7% of the analysis time on average, but it has to be performed over the entire application to decide to classify predicates. It must be considered for future work in order to optimize the number of successfully analyzed applications. To understand the general scheme in which the logic bombs, even false positives, are triggered we extracted for each

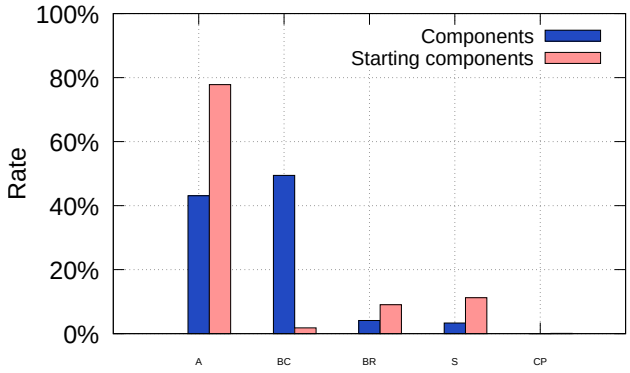


Figure 5.6: Rate of components (A: Activities, BC: BasicClass, BR: BroadcastReceivers, S: Services, CP: ContentProviders)

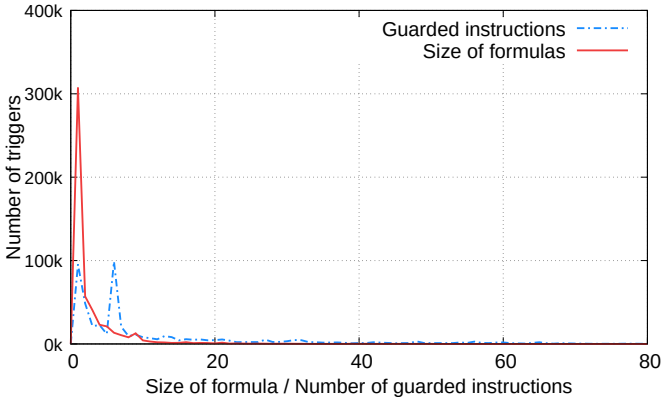


Figure 5.7: Size of logical formula and count of guarded instructions

detected trigger the type of component in which the method triggering the logic bomb is located as well as the component where the call stack starts for reaching this method, referred to as starting component.

In Figure 5.6 we can see that in a large number of cases, components containing the method triggering the logic bomb are non-Android classes (49.44%).

Also, 43.1% are located in `Activities`, meaning that the trigger can also be directly embedded in the user code interface. It makes sense since many applications use time-related triggers for user interfaces (e.g., games).

If we take into account the starting components, it becomes more evident. In fact, almost 80% of the starting components are `Activites`. Many of these are likely to call a method of another class to trigger the logic bomb. Another interesting fact in starting components is that, despite the low proportion, the triggering process often starts in a `BroadcastReceiver` or a `Service`. `BroadcastReveivers` are, in this case, mostly linked to SMS-related triggers. Regarding `Services`, we can assume that this method will likely be used to monitor the device and trigger code at the right moment.

We also extracted two other features to understand the form of the check when detected. We wanted to know if the intra-procedural logic formula extracted during the analysis was complex or not in the general case. We can see in Figure 5.7 that in the majority of the cases, there is only one predicate in the formula, which means that the triggered behavior, in the case of this particular analysis, is generally isolated, not part of a multiple branch decision.

Furthermore, the density of instructions dominated by a trigger is interesting to study.

VT	>0	>10	>20	>30	>40	>50
Apps	29 829	15 861	7919	1237	55	1
FP Rate	17.2%	20.6%	22.6%	24.2%	24.5%	24.6%

Table 5.2: VirusTotal (VT) detection rate of TSOpen flagged applications (October 2019).

Indeed, we can see that most of cases, the number of guarded instructions by a trigger is less than 10 (Jimple instructions). As the number of instructions is small, we can assume that those instructions represent different calls to other classes' methods to perform actions. This assumption correlates with the fact that in most cases, the component in which is the trigger is a basic class (see Figure 5.6), that is to say, a non-Android component class. In fact, in 55.29% of the cases, one of the instructions is a method call, which confirms our previous data records of Figure 5.6.

To retrieve the rate of false positives among the 99 651 detected applications, we based ourselves on VirusTotal [55]. However, the VirusTotal score is challenging to trust for qualifying an application as malware. That is why we decided to classify these applications by detection rate. Table 5.2 shows that the rate of false positives reaches a lower bound of 17.2% and an upper bound of 24.6%.

We applied TSOpen on 508 122 Android applications with a success rate of 79.9%. Our experimentations show that the approach scales on large datasets. However, it also shows that the approach has a high false positive rate of 17% which would require much manual work (which the automated analysis was trying to prevent).

5.3.2 Parameters that impact the false positive rate

The conclusion of RQ1 is surprising since we do not reach the false positive rate of the literature (0.3%). Thus, in this research question, we identify the main parameters that could significantly impact the false positive rate. Since we run many analyses, we cannot use the massive dataset of RQ1. We thus build a new smaller dataset.

In order to build it, we operated as described in the literature. That is to say, we only considered benign applications from Google Play using the minimum score given by VirusTotal [55]. For this, we, again, used the Androzoo dataset [46]. Then, we analyzed the applications to check whether they contained the permission `android.permission.RECEIVE_SMS`, use a location API or a time/date library.

Similarly to the literature, we selected 5803 time-related applications, 4135 location-related applications, and 1400 SMS-related applications. We ended up with a total of 11 338 unique benign applications.

Control Experiment

The control experiment, in which we do not change any parameter, has been conducted in the same context with the timeout set to 1 hour per app. Our analysis was able to successfully analyze 7297 applications out of 11 338 (i.e., 64.4%) with an average of 24 seconds per application. A success means that the analysis for an application did not reach the timeout nor crashed.

The analysis found 9535 suspicious triggers, 4824 applications with a suspicious check, 3636 applications with suspicious triggered behavior and 3099 applications after post-filters (see Table 5.8 for more information) yielding a false positive rate of 27.3%.

On a dataset with two orders of magnitude smaller than in RQ1, we find that the false positive rate still reaches a high value of more than 27%.

	Original results	Timeout 2h	Timeout 3h	Symbolic values filter	Package filter
# apps analyzed	7297	9884	9897	9880	10133
Mean time of analysis	24	141.2s	146.5s	128.6s	13.2s
# of suspicious triggers	5391	7724	7727	1033	83
# of apps with triggers	1701 (23.3%)	2373 (20.9%)	2376 (21%)	381 (3.4%)	31 (0.3%)

Table 5.3: Experimental results with Timeout variation (cols. 2 and 3), Symbolic Filter (col. 4), Package Filter (col. 5).

Sensitive Methods	Occurences	Percentage	false positive rate induced
TextView.setText	688	12.8%	1.6%
ConnectivityManager.getActiveNetworkInfo	600	11.1%	1.2%
File.<init>	544	10.1%	1.2%
Log.v	436	8.1%	1.6%
URL.openConnection	361	6.7%	1.3%
ContextWrapper.startActivity	361	6.7%	1.7%
Location.getLatitude	280	5.2%	0.9%
Log.println	241	4.5%	1.0%
Activity.finish	186	3.5%	0.9%
NotificationManager.notify	163	3.0%	0.6%
File.mkdirs	118	2.2%	0.5%
Handler.sendEmptyMessageDelayed	112	2.1%	1.3%
Handler.sendEmptyMessage	90	1.7%	0.6%
Handler.sendMessage	85	1.6%	0.6%
TelephonyManager.getDeviceId	77	1.4%	0.4%

Table 5.4: Top 15 sensitive methods considered order by number of occurrences in the defaul experiment of Section 5.3.2

Sensitive Methods Filter

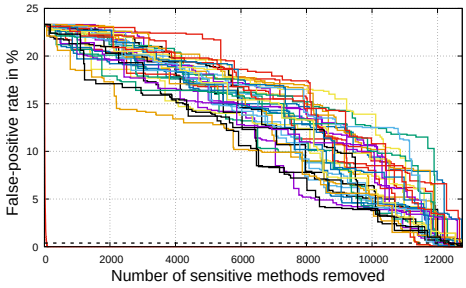
In this experiment, we randomly remove methods in the list of sensitive methods, one after the other, to observe the impact on the false positive rate. We perform this experiment 32 times to see if the results converge. Figures 5.8a shows the results of this experiment. Each curve represents an experiment. We see that in order to reach a low false positive rate (e.g., 0.38%, represented by the dotted line), we have to remove, on average, more than 11 500 methods ($> 90\%$) from the list of sensitive methods, which will be missed during the analysis.

We can also see a curve diving fast on the leftmost side of the graph of Figure 5.8a. It represents the same filter for which the most used sensitive methods are removed first. The sensitives methods are ordered by their occurrence in logic bombs based on the results of the control experiment in Section 5.3.2.

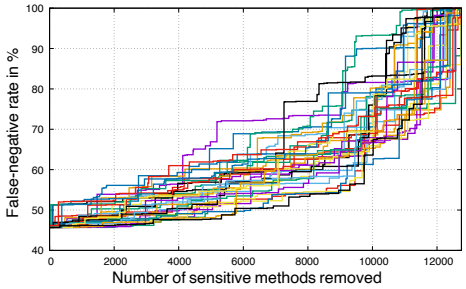
We observe that to reach a low false positive rate represented by the dotted line, removing the 68 most-used methods is enough. This means a concentrated number of sensitive methods are used to qualify a trigger as a logic bomb. Those methods mostly allow one to read device information, write into logs/files, and communicate with the external world.

We provide in Table 5.4 the list of sensitive methods, each present in at least 1% of logic bombs detected in the control experiment of Section 5.3.2. Those 15 methods represent 80.7% of the total of potential logic bombs yielded by our tool. Although some of these methods can be omitted from a definitive list, others like `TelephonyManager.getDeviceId`, which is considered sensitive as it can be leaked and deliver information to the attacker, have to appear in the list of sensitive methods considered. This method alone corresponds to 0.4% of the rate of false positives. We observe that each method in the list can considerably impact the false positive rate.

To measure the rate of false negatives, one needs a control dataset, i.e., a ground truth,



(a) Evolution of the false positive rate.



(b) Evolution of the false negative rate.

Figure 5.8: Evolution of the false positive and false negative rates in function of the number of sensitive methods randomly removed from the list of sensitive methods considered.

which we do not have. In our study, we use a dataset containing only malicious apps to check the number of apps that would not be detected anymore by removing sensitive methods, which we called false negatives. As before, we randomly remove methods from the list of sensitive methods. Figure 5.8b details our findings. We can first see that the false negative rate starts from 45.7%, then we can see that removing the sensitive method increases the false negative rate (while decreasing the false positive rate, see Figure 5.8a). We have seen previously that removing about 11 500 sensitive methods could be helpful to reach a low false positive rate close to 0.3%. In Figure 5.8b, we can see that doing so would set the false negative rate between 70% and 95%, which would be unacceptable for detecting malicious applications.

Changing the list of sensitive methods can significantly impact the false positive rate and the false negative rate, at least up to two orders of magnitude.

Trigger Filter

In this experiment, we modify TSOpen in order not to take into account any potential trigger if the values retrieved during the symbolic execution attributed to the test were purely symbolic or unknown. In Table 5.3 we can see that the number of suspicious triggers drops to 1033 and the number of applications with suspicious triggers to 381. This minor change produces results with a factor 5 change regarding the detection rate. Also, it allows our tool to get a false positive rate close to 3.4%. Unfortunately, the analysis misses all logic bombs where triggers are derived from purely symbolic values.

Using the trigger filter can have a significant impact on the false positive rate.

Package Filter

In this experiment, we modify TSOpen in order to only take into account methods that are in the same package as the application under analysis. This filter is stronger than the library filter of Section 5.3.2 as it constrains more the analysis. The results of Table 5.3 shows that the number of methods analyzed is significantly reduced. Indeed, the time taken for the analysis is shallow compared to the other analyses. Also, the number of triggers yielded by TSOpen reaches 83 in 31 different apps. The rate of false positive is almost equal to the original one. This heuristic has a significant drawback, and an attacker could easily bypass this filter by changing the package name of classes implementing the triggered behavior. Unfortunately, the analysis does not take into account all the code outside of the package. In some applications this accounts for more than 93% of the code.

	Before Lib filter	After Lib filter
# Apps w/ LB	3099	1701 (−45.1%)
# Suspicious LB	9535	5391 (−43.5%)
# LB per App	3.1	3.2
# Time-related	9099	5034 (−44.6%)
# SMS-related	132	117 (−11.3%)
# Location-related	304	240 (−21%)

Table 5.5: Comparison between TSOpen's results before and after filtering common libraries (LB: Logic Bomb)

Using the package filter can have a significant impact on the false positive rate.

Library Filter

In this experiment, we filter out well-known libraries in order to remove noise from the results. For this, we used a list that was made in a study about common libraries [56] used in Android applications. We manually analyzed 35 of them and confirmed that they contain only false positives.

After having filtered common libraries from the triggers found beforehand, our results reveal that a scaling approach with this analysis would still not be conceivable concerning the still high number of triggers detected. Indeed, Table 5.5 shows that even with a reduction of 43.5% of the number of suspicious triggers, there are 5391 suspicious triggers. Also, the number of applications flagged as containing a logic bomb goes from 3099 to 1701, a reduction of 45.1% for the tool but still greater by two orders of magnitude compared to the state of the art. It means that among 11 338 unique benign applications there are potentially 1701 false positives (23.3%).

Note that, despite being conservative, the false positive rate calculated during this experiment is obtained by counting the number of benign applications flagged by our tool containing a logic bomb. This can be explained since a logic bomb necessarily contains malicious code, otherwise, it is triggered behavior. We acknowledge that even being relatively free from malicious applications, picking applications from *Google Play* is not sufficient to qualify the dataset's applications as benign. Nevertheless, we use the same evaluation process to stay in line with the literature.

The majority of detected triggers filtered by the common libraries are time-related triggers. Out of the 4144 suspicious triggers filtered, 4065 (98.1%) are time-related whereas only 15 (0.36%) are SMS-related and 64 (1.54%) are location-related. It shows that common libraries make great use of time-related triggers. Besides, we have already said that suspicious time-related triggers definition was not narrow enough to detect them compared to SMS-related and location-related. We can say that even with an efficient library filter, time-related triggers are still commonly used in benign applications.

The library filter does not significantly impact the false positive rate.

Different list of sensitive methods

To build the list of sensitive methods, we reused the results of Pscout [51] and SuSi [52], as in the literature. The constructed list contains 12 755 methods. Nevertheless, we have seen that we obtain a high false positive rate with this list. That is why we decided to verify the impact if we were to use another, shorter list of sensitive methods.

# apps analyzed	8285
Mean time of analysis	21.1s
# of suspicious triggers	2855
# of apps with triggers	956 (11.5%)

Table 5.6: Experimental results with different list of sensitive methods.

We started from the premise that a permission-based method is not necessarily sensitive. Therefore, we used a list of sink methods from FLOWDROID [5] as they can leak data, which is considered sensitive. The new list features 130 methods.

Nevertheless, the number of triggers flagged by our tool (after re-running the experiment) stays relatively high, reaching 2855. They are distributed in 956 applications (11.5%, see Table 5.6). The tool cannot differentiate between malicious and benign behavior even with a reduced list of methods considered for the control dependency step. This shows the need for a more in-depth analysis of the guarded behavior of the triggers.

Using a reduced list of sensitive methods which are all involved in data leaks does not have a significant impact on the false positive rate.

Concept Drift

Differences in results between TSOPEN and existing experiments of the literature could be due to *concept drift* [47], i.e., the fact that applications used in the experiments of existing papers are older than the ones used in our study. We launched experiments on multiple datasets of 10k apps from 2013 to 2016 and have the following results for the false positive rate: 2013: 18.5%, 2014: 15.7%, 2015: 21.1% and 2016: 22.6%. We observe no significant impact on the results.

Variation in the application release dates does not significantly impact the false positive rate.

Timeout variation

Experiments in the literature could have been conducted a couple of years ago. To simulate the hardware available at the time, we performed the experiments with shorter timeouts. We launched experiments on multiple datasets of 10k applications and have the following results for the false positive rate: 30min: 16.1%, 15min: 15.8%, and 5min: 15,7%. We observe no significant impact on the result.

Reducing the timeout does not significantly impact the false positive rate.

Call graph construction algorithm

The literature might be imprecise and might not always provide all information regarding the implementation of the tool they developed. Mostly, a crucial part of performing interprocedural analyses is the call graph construction algorithm. Therefore, as we do not always know which call graph algorithm is used, we renewed our previous experiment by varying the algorithm. For this, we used the following call graph construction algorithms: SPARK [50], CHA [57], RTA [58] and VTA [59].

Table 5.7 reveals our experimental findings. First, we can see that none of these algorithms allow us to get a low false positive rate close to 0.3%. It can be deduced that having the correct algorithm will not suffice for a perfect implementation. Second, even though the results with the Spark algorithm and VTA algorithm are close, we can see that

changing the call graph construction algorithm leads to different results (i.e., false positive rates of 12.6% for VTA, 11.4% for CHA, 13.6% for RTA, and 11.5% for SPARK).

Besides, we run the same experiment by increasing the timeout to 2h and 3h. Table 5.3 shows the results of these experiments. We can see that there is almost no difference between a timeout of 2h and 3h. However, considering the initial timeout of 1h, the results are different here. Indeed, the timeout of 1h yielded 1701 applications with triggers, against 2373 and 2376, respectively 2h and 3h. Likewise, the number of triggers detected increases from 5391 with 1h to 7724 and 7727 with respectively 2h and 3h.

Changing the call graph construction algorithm does not significantly impact the false positive rate.

We have experimentally seen that minor changes in the implementation can have an important impact on the results. Using heuristics allows the approach to get a false positive rate similar to the literature (0.3%). However, this result has a significant impact on the recall, the false negative rate being raised between 70% and 95%.

5.3.3 Can logic bomb detection help localize malicious code?

This is by far the most interesting question for this research area. Indeed, detecting malicious code is a difficult problem per se, that is why if this approach could help in this direction, it could be promising. In fact, TSOpen's approach is efficient for this purpose for applications taken individually. Indeed, we manually analyzed 200 apps, and we were able to locate quickly (i.e., in less than 2 minutes on average) the malicious code with the results yielded by TSOpen. When a true positive is encountered, we can directly inspect the method in which the logic bomb is. Consequently, we had malicious code at hand.

We could locate/track the malicious code during our numerous manual analyses. The condition of the logic bomb playing the role of the malicious code entry point.

5.3.4 Behavior similarity between goodware and malware

In this section, we analyze randomly chosen malicious and benign Android applications containing a trigger.

Malicious The first malicious application we present is called "LittlePhoto"[1] and allows a person with malicious intents to install third party applications and receive information about the device via HTTP by sending an SMS with `"$$@@&&$$"` or `"$$@@&&@@"` as the content. It can be viewed as a targeted attack which uses a logic bomb detected as `#sms/#body.equals('$$@@&&$$')`. This kind of SMS-related logic bomb is usual in remote administration tool (RAT) or SMS-based backdoors.

The second one is called "com.allen.mp"[2] and this time relies on a time-related triggered behavior : `"#now cmp 14400000L"`. After decompiling the application and analyzing it (see Listing 5.1 for an example of the code), we found that it checks if there is a ten days period between the current time and a pre-defined value. If the condition is satisfied, the application retrieves information about the type of operating system, the version of the Android framework, the model of the device, the number of the device, the operator, the type of network, and information about the storage. Then it sends all this information to a C&C server: `"http://search.gongfu-android.com:8511/search/sayhi.php"`.

[1] 9c92c2279a33de01561ce775c8beee9bbb58895a1f632d19f41ac2b286e12bb2
[2] 54f3c7f4a79184886e8a85a743f31743a0218ae9cc2be2a5e72c6ede33a4e66e

```
public void onCreate() {
    long j = sharedPreferences.getLong("start", 0);
    long currentTimeMillis = System.currentTimeMillis();
    if (currentTimeMillis - j < 14400000) {stopSelf();}
    else {
        if (Utils.isConnected(this)) {doSearchReport();}
        getPermission();
        provideService();
    }
}
```

Listing 5.1: Time-bomb in com.allen.mp application (simplified)

```
public void onReceive(Context context, Intent intent) {
    SmsMessage[] smsMessageArr = (SmsMessage[]) null;
    String str = getMessageBody(smsMessageArr);
    String code = "abacdacdcadcdacdacadcacd--ca--c-da-dca-cda-c-ac-a-c-adc-a-c-a";
    if (str.equals("zebinjo")) {
        CrnacVibrator crnacVibrator = new CrnacVibrator();
        crnacVibrator.odgovor(code);
        abortBroadcast();
    }
}
```

Listing 5.2: Exam tool decompiled (simplified)

Benign The first benign application we present is called "Exam Tool"[3]. It allows students to cheat during an exam without looking at the phone but get the answer from a friend by the number of vibrations the phone would make depending on the received SMS. TSOpen flagged this application with this logic bomb : `#sms/#body.equals("zebinjo")`, it is clear that it is suspicious according to the definition of a logic bomb. However, is it malicious? Even if it calls a method categorized as sensitive (on the list of sensitive methods), the answer is no. Indeed, depending on the SMS received, the phone will vibrate according to a particular protocol defined in the application. But, a hard-coded response code is in the `onReceive` method of the `BroadcastReceiver` receiving the SMS. And when receiving exactly the string "zebinjo" it will trigger the vibration according to the following scheme : "abacdacdcadcdacdacadcacd–ca–c-da-dca-cda-c-ac-a-c-adc-a-c-a-dca-cac-a-dc-ad", the code can be seen in Listing 5.2.

This example shows why TSOpen's approach cannot be entirely reliable because of the trigger's focus and not the triggered behavior itself. It implies detecting malicious code, i.e., the problem reduces to detecting it.

Another example of SMS-related triggered behavior is the family of tracker applications allowing users to get information on a device using SMS commands remotely. The application using this kind of triggered behavior we are going to analyze is "MyCarTracks"[4]. The trigger condition discovered by TSOpen is `#sms/#body.startsWith("GETPOS")` visible in Listing 5.3.

The next benign application named "TrackMe"[5] makes the use of a time-related trigger-based behavior, which is part of the definition of TSOpen's approach but is clearly legitimate. The detected logic bomb is `#now cmp 15L` which is a comparison between the current date and a numeric value of 15, see Listing 5.4 for the code. It appears that it is simply a check performed to verify the validity of the trial version of the application.

[3] c373d79960eadfd34fe56ad67051e2d39536c393513667b9d3c05bcd601dd874
[4] 17bf1d8afc22a40681f833cf442732795f134e3a4b287fd9f99d21db3fa07a81
[5] a22374fd3a2f6a91ab15d06be29b0a2134a1756d3aa8afe5d7cac8470b418c8d

```
if (intent.getAction() != null) {
    Bundle extras = intent.getExtras();
    if (extras != null) {
        Object[] objArr = (Object[]) extras.get("pdus");
        SmsMessage smsMessage =  SmsMessage.createFromPdu((byte[]) objArr[0]);
        if (objArr.length >= 0) {
            String body = smsMessage.getMessageBody().toString().toUpperCase();
            if (body != null && body.startsWith("GETPOS")){sendPosition();}
        }
    }
}
```

Listing 5.3: My Car Tracks decompiled (simplified)

```
if ((System.currentTimeMillis() - Config.Review) / 86400000 < 15) {
  builder = new Builder(MainActivity.this);
  builder.setTitle("Trial expired");
  builder.setPositiveButton("Yes", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialogInterface, int i) {
      Intent intent = new Intent(MainActivity.this, BillingActivity.class);
      MainActivity.this.startActivity(intent);
    }
  });
}
```

Listing 5.4: Track Me application decompiled (simplified)

	CHA	**RTA**	**VTA**
# apps analyzed	2414	3724	7605
Mean time of analysis	37.4s	39.1s	37.4s
# of suspicious triggers	817	1887	2925
# of apps with triggers	275 (11.4%)	506 (13.6%)	957 (12.6%)

Table 5.7: Experimental results with different call graph construction algorithms and a reduced list of sensitive methods considered. (CHA: Class Hierarchy Analysis, RTA: Rapid Type Analysis, VTA: Variable Type Analyis)

```
SimpleDateFormat sdf = new SimpleDateFormat("MMddyyyy");
StringBuilder date = new StringBuilder(sdf.format(new Date()));
Object[] obj = (Object[]) intent.getExtras().get("pdus")
SmsMessage sms = SmsMessage.createFromPdu((byte[]) (obj)[0]);
if (date.toString().matches("05212011")) {
  int nextInt = new Random().nextInt(4) + 1;
  if (sms.getMessageBody().matches("health")) {
    deleteContent();
  }
  String s1 = "Cannot talk right now, the world is about to end";
  String s2 = "Jesus is way over due for a come back";
  String s3 = "Its the Raptures,praise Jesus";
  String s4 = "...";
  String[] strArray = new String[]{s1, s2, s3, s4};
  SmsManager sm = SmsManager.getDefault();
  sm.sendTextMessage(sms.getOriginatingAddress(), null, strArray[nextInt], null, null);
}
```

Listing 5.5: Holy Colbert application decompiled (simplified)

Our manual investigations have shown that benign and malicious applications can use the same code for benign and malicious behavior. Therefore, in this case, the problem of qualifying malicious code remains.

5.3.5 TriggerScope reproducibility

The experiments have been conducted on two datasets: the first is a dataset of malicious applications, and the second is a dataset of benign applications. To faithfully reproduce the experiments, we wanted to use the datasets of the experiments in the original paper. Unfortunately, the list of benign applications has been lost. Hence, we created a new dataset which has the same properties as the original dataset. Concerning malicious applications, 3 out of the 14 considered in their experiments were shared with us.

Malicious applications

We have executed TSOpen over the three malicious applications TriggerScope's authors were willing to share to check if the same logic bombs described in their paper could be found. The first one is called *Holy Colbert*, the second one comes from the *Zitmo* malware family, and the last one is the *RCSAndroid* malware.

First, when executing TSOpen over the application called "Holy Colbert" coming from the so-called MalGenome dataset [60] we effectively find the same time-bomb as they did, but not only, we also discovered an SMS-bomb, see Listing 5.5 for more details. The SMS-bomb revealed by our tool is `#sms/#body.matches("health")` which represents a suspicious narrow check against the body of an incoming SMS. It is triggered if the time-bomb is satisfied, meaning at a specific date, here the May, 21^{st} 2011. It triggers the deletion of data through the content resolver. This implies that our implementation is not entirely identical to the original. We do not claim that this additional finding makes our implementation more precise because it may introduce more false positives in other applications.

Finally, regarding the "Zitmo" and "RCSAndroid" malicious applications they provided us, TSOpen was able to extract the same logic bombs as TriggerScope.

Domain	# Apps	# w/ SC	# w/ STB	# After PF
Time	2967 (4950)	1719 (302)	1263 (30)	1094 (10)
Location	3305 (3430)	2366 (71)	1817 (23)	1516 (8)
SMS	1025 (1138)	739 (89)	556 (64)	489 (17)
Total	7297 (9518)	4824 (462)	3636 (117)	3099 (35)

Table 5.8: Result of our analysis on the 11 338 benign applications. The values in parenthesis represent TriggerScope's results for the original dataset. (SC: Suspicious Checks, STB: Suspicious Triggered Behavior, PF: Post-Filters).

Reproducibility of TriggerScope's results

In this section, we further investigate the discrepancies between TSOpen's results and TriggerScope's.

In the original paper, the authors had a total of 9582 unique benign applications due to overlap between categories. We made the list of the 11 338 applications public in the project repository for reproducibility purposes.

First, we observe in Table 5.8 a considerable difference between our analysis and TriggerScope's: while TSOpen identified 3099 suspicious apps, TriggerScope identified only 35 [6]. While it is true that the two datasets are different, we did not expect to find a two-order of magnitude difference between the results.

Second, we extracted different features from each application and each potential suspicious check to understand and verify our results. Among them, we retrieved the class containing the suspicious check and the method in which it appears, and the sensitive method invoked to flag it. Only 20 methods in the list of sensitive methods represent 89% of the sensitive methods considered to flag the suspicious checks. The list of sensitive methods that we used might explain why we have such a difference in our results. Nevertheless, it cannot explain the gap alone because other factors could impact the results. Consequently, we also analyzed the classes and methods containing suspicious triggers to verify if some distinct pattern might emerge.

We found 3165 different combinations of class/method among the 9535 suspicious triggers without any combination being overly represented. We manually analyzed the most used of them (i.e., the first 35) to verify if they were logic bombs. They seem to enter the definition of a logic bomb according to the paper in question, but they are not. In Listing 5.6 we can see an example in the *card.io* library, which executes some code if some time has elapsed, then it executes the method called `android.hardware.Camera.Open()` which is considered suspicious in the list of sensitive methods. We chose this example to emphasize that most of them are based on time-related triggers. Better, we found that within the 9535 logic bombs found (among the 3099 applications flagged), only 3.1% were location-related and 1.4% were SMS-related.

We also note that most of the suspicious triggers (43.5%) are part of a library used in applications. Manual analyses revealed that they are not logic bombs, thus introducing noise in the analysis. We observe that no filter mechanism is mentioned in TriggerScope's paper. We contacted the authors, but we could not get the information on whether a filter was used or not in their implementation. According to our results, to reach such a low rate of false positives (0.38%), at least a library filter has to be used to rule out repeated false positives.

The reader may have noticed the structure of the previous logic bombs, i.e., nested conditions. It raises the question of whether this type of structure is often used in trigger-

[6] Unfortunately, TriggerScope authors were unable to run their tool on our dataset to compare the results, so we reused the results of their paper on their original dataset to compare our results.

```
1  private Camera connectToCamera(int checkInterval, int maxTimeout) {
2    long start = System.currentTimeMillis();
3    if (this.useCamera) {
4      do {
5        try {
6          return Camera.open();
7        }catch (RuntimeException e) {
8          // do something
9        }
10     } while (System.currentTimeMillis() - start < maxTimeout);
11   }
12   return null;
13 }
```

Listing 5.6: Trigger in io.card.payment.CardScanner class of card.io library (simplified). The `Camera.open()` method (on the list of sensitive methods) is triggered -not only- under the condition triggered by the *while* instruction.

based behavior. Also, we want to verify if considering nested conditions could have been treated as a single logic bomb by TriggerScope developers. Therefore, it could explain the gap between our results and theirs. We measured this and found that only 16.38% of detected triggers have a nested structure. According to this number, we can conclude that it does not impact the conclusion when comparing TriggerScope and TSOpen.

We were able to construct a dataset with the same properties as the one used in the original paper. However, TSOpen yielded a high false positive rate by detecting 3099 potential apps with logic bombs ($> 27\%$).
We see the importance of having the original list for reproducing this experiment, as it can significantly impact the false positive rate. Therefore, with the information provided to us and the description of the approach made in the original paper, we conclude that the approach might be used in a realistic setting to detect logic bombs.

5.4 Discussions

Trigger types. Only three trigger types have been modeled, which is not representative of logic bombs, though expanding it to other types would be easy. Regarding other types of logic bombs, we recently found that a new banking Trojan named "Cerberus" made clever use of the accelerometer sensor for monitoring the device [61]. Indeed, it is based on the assumption that a real person would move with their device, hence changing the step counter's values. Only if this condition is satisfied would the malicious code be triggered.

In a recent analysis, Stone found a malicious application using multiple evading techniques [62]. The malware will first check if the device has a Bluetooth adapter and a name, which is important as emulators use default names. Then it would check if the device has a sensor and verify the content of `/proc/cpuinfo` to find both *intel* and *amd* strings. As most devices use ARM processors, those strings should not appear. It also checks the appearance of any *Bluestacks* files, an emulator solution, and other emulation detections. Finally, this application would deliberately throw an exception and check the content to find any matching string showing an emulator's existence.

Predicate Minimization. The next limitation lies in the fact that predicate recovery and predicate minimization are performed systematically, which increases the probability of running into a complicated formula for which the minimization step would never end. Besides, this step is responsible for 22.2% of the analysis time and accountable, in 35.3% of the cases, for reaching the timeout of the analysis. Unlike the symbolic execution step,

which is responsible for the most significant part of an application's analysis time (61.7%), the path predicate recovery and minimization are unnecessary for the entire application. Indeed, the symbolic execution phase is necessary to decide on the suspiciousness of conditions. A countermeasure would be to locate interesting checks and then perform the full path predicate recovery and minimization.

Maliciousness. The most critical weakness of TriggerScope approach is the control dependency step, which compares method calls dominated by suspicious triggers with a list of sensitives methods. This phase requires more attention as it is used to qualify the maliciousness of a condition. Indeed, a suspicious check can be harmless due to the same usage benign and malicious applications make of trigger-based behavior. Nevertheless, we recognize the difficulty of this step, given the lack of a formal definition of malware. Despite having considerable resources, major companies also realize the difficulty of automatically qualifying malicious code, e.g., Google still accepts malicious applications in its PlayStore [63].

Implementation Errors. Even though we reproduced faithfully the approach described in TriggerScope paper and reused available and well-tested state-of-the-art code when possible, we are not immune from implementation errors.

Implementation Unknowns. Given the details in the original paper, we cannot reproduce the results. Therefore, we tried to vary parameters and implementation details to get as close as TriggerScope's results. However, it is challenging to test all the combinations of implementation/parameters to get the original results.

5.5 Related Work

In 2008, Brumley & al. [30] developed MineSweeper which is an interesting approach to assist an analyst. Their solution worked directly at the binary level of an executable application. Their goal was to uncover trigger-based behavior by constructing conditional paths and input values to execute the application. The next step was to ask a solver whether the path is feasible or not. If not, they would not explore this path. On the contrary, they would explore this path and ask the solver to construct -if possible- input values to satisfy the formula. They then execute the application with the computed trigger input values to inspect the behavior. If a malicious behavior were encountered, they would know the conditional path leading to this behavior, thus detecting if a logic bomb exists. They conducted their experiments on four real-world applications and succeeded in finding trigger-based behavior in less than 30 minutes per application with less than 14 potential logic bombs per application. Unlike TSOpen, the process is not entirely static nor fully automatic and requires a human to infer the logic bomb.

Four years later, Zheng & al. [14] focused on finding a user interface-based trigger that could be used to hide malicious code from traditional analysis in Android applications. They construct the FCG (Function Call Graph) to retrieve call paths to sensitive Android APIs. The next step is constructing the ACG (Activity Call Graph) to have the relationship between the application activities, i.e., how to go from one activity to another via user interface methods. Having those details, they run the application by triggering user interface elements to go to the sensitive activity, which calls a sensitive API and monitors the behavior to check if anything suspicious happens. They can deduce if the user interface triggering process is used to activate malicious code. Their approach is not generic and focuses on one single type of logic bomb. Also, we note the use of dynamic analysis of their approach.

Pan & al. [64] presented a new machine-learning based technique to detect *Hidden Sensitive Operations* in Android apps. They do not specifically focus on malicious behavior, contrary to TriggerScope's approach, which targets malicious activities. Their

approach comprises a pre-processing part where lightweight data flow and control flow analysis are performed to extract a condition-path graph. This latter is then used to extract features that will feed the SVM classification. Doing so, HSOMiner performs a precision of 98.4% (based on 125 randomly chosen apps from a set of 63 372) and a recall of 94%. Though the approach is interesting (SVM is resistant to overfitting), it does not fit the goal of detecting logic bombs hidden in Android apps.

In 2017, Papp & al. tried to work directly at the source code level to detect trigger-based behavior in legitimate applications [44]. Their goal is not to work on a malicious application. They want to emphasize triggered behavior or backdoor behavior in legitimate open-source applications. For this purpose, they use KLEE [65] to perform mixed concrete and symbolic execution (also called concolic execution). However, they must first instrument the application to add specific library calls to use these existing tools. Then they execute the concolic execution based on the result and generate different test cases. Out of these test cases, some can be highlighted by their program to be verified by an analyst to check whether it is a trigger-based behavior or not. Though it is promising for a semi-automatic detection tool, their approach can take a significant amount of time to generate test cases and lead to many false negatives.

We now present a more advanced and promising method developed by Bello and Pistoia [66]. Their approach aims at exposing evading techniques that sophisticated malware use. Nevertheless, as we will see, they do not stop at the detection. Their work is divided into three parts. The first is detecting *evasion point candidate* using information flow analysis. They use the notion of source and sink, that is to say, detecting information going from a source and going to a sink. Once detected, they step into stage two, simply the Java bytecode instrumentation to force the untaken branch during the following analysis. The last stage is executing the instrumented application in a controlled environment to monitor the malicious code's behavior. According to the authors, their approach is unsound but a stepping stone toward detecting new malware behavior. The first stage of their work is related to our work, except that they follow the flow of fingerprinting methods to branches.

Logic bombs can be used for venerable purposes, indeed in a recent study, Zeng & al. [67] presented an approach to detect repackaging using triggers. Indeed, their approach consists of instrumenting a legitimate app that could be repackaged by an attacker and adding an instruction to make the app "repackage-proof". They introduce cryptographically obfuscated trigger conditions that, when triggered, can detect if the program has undergone a repackaging process. The term "logic bomb" specifically means triggering malicious code under specific circumstances. However, the code triggered is not malicious but preventive. Therefore, although they use the same mechanism as malware developers, the authors should talk about *Hidden Preventive Code*. Nonetheless, their approach is resilient since we assume the condition has been detected, which is already a challenging problem. One cannot resolve the condition due to its cryptographic properties (using hash functions). Therefore the guarded code cannot be decrypted and executed. Inserting checks in legitimate apps is promising for protecting Android apps from malware developers and has been well studied in the literature [68].

5.6 Summary

In this work, we have implemented TSOpen, the first open-source version of the state-of-the-art approach, TriggerScope, for detecting logic bombs in Android applications. We first conducted a large-scale analysis over a set of more than 500 000 Android applications and observed that the approach scales.

However, the approach is not appropriate for automatically detecting logic bombs

because the false positives rate of 17% is too high. We conducted multiple experiments on the approach's parameters to understand the impact on the false positive rate and identify that a low false positive rate could be reached but at the expense, for instance, of missing a large number of sensitive methods. That is to say, the approach with a low false positive rate misses a large number of logic bombs. We experimentally show that TSOpen's approach might not be usable in a realistic setting to detect logic bombs with the original paper's information.

Moreover, we have seen that TSOpen's approach is insufficient to detect logic bombs because benign and malicious apps can use the same code for benign and malicious behavior. This is a direct consequence of the lack of a formal definition of malware. We empirically show that using TSOpen's approach, *trigger analysis* is insufficient to detect logic bombs. Indeed, dissociating the trigger condition and the guarded behavior produces false positives. Thus, an analyst is necessary to verify the behavior. Nevertheless, when manually inspecting malicious applications containing logic bombs provided by TSOpen, we could quickly verify if the triggered code was malicious. We did not have to search through all the code, which saved us a lot of time. Hence, TSOpen's approach seems promising to locate the malicious code using logic bombs in reverse-engineered applications.

In a nutshell, our work shows that, even though TSOpen's approach is interesting, it is not suitable for automatically detecting logic bombs. Indeed, the control dependency step is not sufficiently representative of malicious behavior. Our results contradict the state of the art by two orders of magnitude regarding the rate of false positives. We have identified several parameters, such as the list of sensitive methods, which could impact the false positive rate on a two-order magnitude. These parameters should be detailed in every paper tackling the challenging task of detecting logic bombs in Android applications. We hope that future publications do not omit this information to make their experiments reproducible.

Chapter 6

Uncovering Suspicious Hidden Sensitive Operations in Android Apps

In this chapter, we propose to investigate Suspicious Hidden Sensitive Operations (SHSOs) as a step toward triaging logic bombs in Android apps. To that end, we develop Difuzer*, a novel hybrid approach that combines static analysis and anomaly detection techniques to uncover SHSOs, which we predict as likely implementations of logic bombs. Concretely,* Difuzer *identifies SHSO entry points using an instrumentation engine and an inter-procedural data flow analysis. Then,* Difuzer *extracts trigger-specific features to characterize SHSOs and leverages One-Class SVM to implement an unsupervised learning model for detecting abnormal triggers. While this approach has proven effective in statically triaging logic bombs among SHSOs in Android apps, it outperforms the current static logic bomb detector* TriggerScope*.*

This chapter is based on our work published in the following research paper:

- **Jordan Samhi**, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Difuzer: Uncovering Suspicious Hidden Sensitive Operations in Android Apps. *In Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (**ICSE**)*. IEEE, 2022, 10.1145/3510003.3510135 [7].

Contents

6.1 Overview

Security and privacy in Android have become paramount given its pervasive use in a wide range of user devices, be it handheld, at home, or in the office [38]. Yet, new threats are regularly discovered, even in the official Google Play app store [69]. Typically, thousands of apps are regularly flagged by antivirus engines. For the year 2020 alone, the AndroZoo [46] repository has collected over 228 000 apps, among which over 10 000 apps are flagged by at least five antivirus engines hosted by VirusTotal. Addressing the spread of malware in app markets is a prime concern for researchers and practitioners. In the last decade, several approaches have been proposed in the literature to automate malware identification. These approaches explore static analysis techniques [2, 4, 17, 44, 70], dynamic execution [12, 71, 14], or a combination of both [29, 31, 30], as well as the use of machine-learning [72, 73].

While the aforementioned techniques have been proven effective on benchmarks, attacks evolve rapidly with increasingly sophisticated evasion techniques. Typically, malware writers rely on code obfuscation [33] to bypass static analyses. To evade detection during dynamic analysis, attackers seek to hide malicious code behind triggering conditions. These are known as *logic bombs*, the triggering conditions being varied. For example, a logic bomb could execute malicious instructions only at a specific time that is not likely to be reached when market maintainers dynamically analyze the software before it is distributed.

Logic bombs can be used for any malicious activity such as adware [74], trojan [75], ransomware [76], spyware [77], etc. [60]. Furthermore, as the trigger and the malicious code are generally independent of the core application code, logic bombs can easily be added in legitimate apps and repackaged for distribution [78, 79, 80, 81]. Therefore, detecting logic bombs is of great importance, especially in mobile devices that carry much personal information. However, due to the undecidable nature of this detection problem in general [82], and the fact that dynamic analyses will likely fail to detect such behaviors [83], analysts explore static analysis based heuristic or machine learning approaches to detect logic bombs.

A logic bomb is characterized by the fact that it implements a hidden sensitive operation. Therefore, recent works addressing logic bombs have focused on identifying Hidden Sensitive Operations (HSOs) as a target [64]. However, not all HSOs are logic bombs. Indeed, an HSO may be neither **intentional** nor **malicious**, while logic bombs always are. In this work, we propose identifying **Suspicious HSOs** (SHSO) towards triaging logic bombs among HSOs. Indeed, we consider that an SHSO is an HSO that is likely to implement a logic bomb. Further note that this study does not attempt to address a binary classification problem of discriminating malware from benign apps (e.g., by using logic bombs as a key criterion of maliciousness). Instead, our ambition is to improve the detection of logic bombs, considered sweet spots for targeting the understanding of malware's malicious behaviors. Indeed, while the literature proposes various approaches for predicting Android apps' maliciousness (i.e., malware detection), the community still seeks to make significant breakthroughs in localizing malicious code parts. Detecting logic bombs thus provides an opportunity to localize and characterize malicious code implemented as hidden sensitive operations.

Recent literature on Android has already approached the problem of detecting sensitive behavior triggered only when certain conditions are met. Such triggers are referred to hereafter as *sensitive triggers*. TriggerScope [17] was proposed as a static analysis tool to detect logic bombs: its analyses are based on heuristics and are thus limited to certain trigger types (i.e., time-related, location-related, and SMS-related triggers). TriggerScope further relies on symbolic execution, which reduces its capacity to scale to massive datasets. Unlike TriggerScope, HsoMiner [64] leverages a supervised learning

approach with engineered features to reveal sensitive triggers. HsoMiner, however, does not specifically target malicious triggers: it flags up to 20% of apps, which makes it inefficient for isolating dangerous triggers in the wild; it also takes on average 13 min/app, which makes it challenging to exploit for large-scale experiments.

HSO triggering conditions are typically implemented by *if statements*. However, a given app code may contain hundreds to thousands of such conditional statements. Therefore, a major challenge in the research around HSO is to reduce the search space for accurately spotting suspicious sensitive triggers. Our core idea towards achieving this ambition is to model specific trigger characteristics to spot SHSOs.

In this work, we propose a novel approach to identify suspicious hidden sensitive operations where we rely on an unsupervised learning technique to perform anomaly detection. We intend to detect suspicious triggers deviating from the normality of the myriads of conditional checks performed in typical apps. To do so, we explore specific trigger/behavior features to guide our detection system toward enumerating truly suspicious triggers and thus refine the search space for uncovering logic bombs. We propose Difuzer, a novel hybrid approach that combines ① code instrumentation to insert particular statements required for taint analysis, ② inter-procedural static taint analysis to find suspicious sensitive triggers, and ③ anomaly detection to reveal *Suspicious Hidden Sensitive Operations* in Android apps.

While the literature includes work [64] that proposed supervised learning techniques for detecting HSOs, Difuzer relies on unsupervised learning to spot "abnormal" triggers. Moreover, to ensure that the model is accurate in detecting suspicious HSOs, Difuzer leverages features specifically engineered to capture semantic properties of maliciousness.

The main contributions of our work are as follows:

- We propose Difuzer, a novel approach to detect SHSOs in Android apps. Difuzer combines code instrumentation, static inter-procedural taint tracking, and anomaly detection techniques.
- We evaluate Difuzer and show its ability to reveal SHSOs with a 99.02% precision in less than 35 seconds on average per app, outperforming previous approaches.
- We demonstrate that the trigger- and behavior-specific features of Difuzer are relevant for triaging logic bombs among HSOs: 29.7% of detected SHSOs is indeed confirmed as logic bombs.
- We compare Difuzer against a state-of-the-art logic bomb detector, TriggerScope: Difuzer reveals more logic bombs than TriggerScope while yielding fewer false positives.
- We further applied Difuzer on a dataset of "benign" apps from Google Play. By analyzing the yielded SHSOs, Difuzer contributed to suspect 8 adware apps, which Google removed from Google Play after we pointed them out.
- We release the Difuzer prototype in open-source:

`https://github.com/JordanSamhi/Difuzer`

6.2 Approach

Goal: With Difuzer, we do not aim at detecting any HSOs, but only suspicious HSOs (SHSOs) for which the likelihood of being logic bombs is high.
Intuition: As shown in previous studies [64], the number of HSOs per app can be large, even in benign apps. This suggests that although HSOs are "sensitive" operations, most of them are legitimate, i.e., they are used to implement common behavior. In contrast,

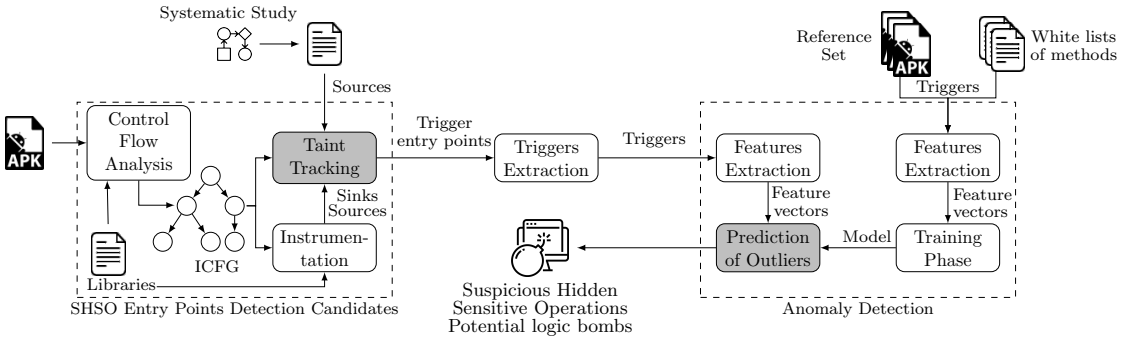


Figure 6.1: Overview of the DIFUZER approach.

logic bombs are rare, especially in benign apps. The idea behind DIFUZER is to use an anomaly detection approach, with specifically designed features, to triage logic bombs among SHSOs.

Overview: In Figure 6.1, we present an overview of our approach, which consists of two main modules: (1) identification of SHSO entry point candidates via control flow analysis, instrumentation, and taint tracking (left dotted block); (2) From these entry points, triggers are extracted, and the second module (right dotted block) extracts specifically designed features fed into an outliers predictor. This predictor is previously trained on a set of reference apps (i.e., apps considered benign) to learn legitimate usages of triggers.

6.2.1 Identifying SHSO candidate entry points

Previous works [84, 85, 86, 87, 12] have shown that specific values, such as system inputs and environments variables, are often used to trigger HSOs. State-of-the-art approaches have thus proposed to check whether the conditions of *if statements* contain these sensitive data. To that end, they rely on symbolic execution [17] or backward data-dependency graphs [64] that could suffer from scalability problems. With DIFUZER, we propose to use taint analysis to track sensitive data values and check if they are involved in conditional expressions.

Taint analysis tools generally track data from sources to sinks. The implementation of FLOWDROID, a popular taint analysis framework for tracking sensitive information, considers sources and sinks at the method level. In our case, however, sinks are fine-grained code locations, which are conditional expressions of *if statements*. This requires for DIFUZER to instrument apps to insert dummy method calls that will ready the apps for analysis by FLOWDROID (cf. Section 6.2.1). Moreover, sources can be method calls or data field accesses. To build the set of sources and sinks, we propose to make a systematic mapping (cf. Section 6.2.1) that explores internal and external system properties and their associated APIs as well as environment variables.

Systematic mapping toward defining sources

As already explained, a first step is to track sensitive values. In this work, these values are derived from particular source methods. Then, if a sensitive value falls into an *if statement*, we consider the condition as a potential SHSO entry point. This section will describe how we gathered a comprehensive list of source methods used for the taint tracking phase. Note that we did not rely on the reference sources list produced by SUSI [52] since it has been shown that most of the methods are inappropriate for tracking sensitive data and lead to a high amount of false positives (e.g., >80%) [88, 89, 90].

In general, decisions on whether to trigger SHSOs or not are taken on system proper-

	Device					
	Internal			External		
	System	**Content**	**Build**	**SIM**	**Internet**	**GPS**
Examples	Sensors, Camera	Call Logs, Contacts	Model, Hardware	Phone call, SMS	Parameters, Content	Latitude, Longitude

Table 6.1: Examples of sensitive sources

ties [84, 86, 62, 64]. Hence, we performed a systematic mapping of the Android framework from SDK version 3 to 30 (versions 1 and 2 were unavailable) to gather a comprehensive list of source methods. In particular, since in the case of Android apps, system properties can be derived from the device's *internal* and *external* properties, we inspect the successive versions of the framework to identify various means to access these properties.

In Table 6.1, we enumerate the different property types (with examples) on which we reasoned to retrieve sensitive sources, which are classically focused on in the literature [84, 86, 62, 64]. We follow a systematic process to perform the retrieval of sources from the given property types: we first extracted patterns from the different ways to access the aforementioned properties. Then, we used those patterns to automatically discover the sensitive sources that we make available to the research community in the DIFUZER project's repository. In the following, we further detail the internal and external properties that we consider.

Internal: In the case of internal properties, a developer can get sensitive information of the device from three main channels: 1) System properties, 2) Content in internal databases, and 3) Information from `BUILD` class (see Table 6.1). In the following, we describe how we obtain a list of sources for those three channels:

① *System properties*: While developing an Android app, developers have access to several useful APIs. In this case, the most interesting is `android.content.Context.getSystemService(java.lang.String)` [91] which returns the system-level handler for a given service. The service is described by a string given as parameter to `getSystemService` method. The `Context` class gives developers access to pre-defined constants (e.g., `SENSOR_SERVICE`).

In fact, every constant contains the name of the service with `"_SERVICE"` appended to it. The return value type of the `getSystemService` method call is derived from the constant name (e.g., `SENSORSERVICE` will give a `SensorManager` [92]) which in turn can be used to get a object whose type is also derived from the constant name (e.g., a SensorManager object can be used to obtain a Sensor object [93]). We used this pattern to compile our list of sensitive sources for the System properties. More specifically, we verify if the class exists in at least one SDK version for each class obtained. If this is the case, we list the methods of the class and keep only the "getter methods", i.e., those starting by "get" or "is" (e.g., methods such as `getId()` or `isWifiEnabled()`).

② *Content in internal databases*: To access databases fields, one has to perform a query which returns a `android.database.Cursor` [94] object. This object is then used to iterate over the result of the query. Hence, to get sensitive source methods related to content in internal databases, we applied the same process as for system properties (i.e., to retrieve the "getter" methods) but on the `Cursor` class.

③ *`Build` class*: The `Build` class [95] allows developers to access information about the current build of the device from its fields. For instance, one can get the brand associated with the device by accessing `Build.BRAND`. Note that our objective is to retrieve a list of source methods. However, the information a developer can get from the `Build` class can only be retrieved from class fields, not method calls. Consequently, in Section 6.2.1, we will explain how we instrument the app under analysis to add method call statements representing `Build` field accesses.

```
1  public void method() {
2    String b = Build.BRAND;
3  + b = BuildClass.getBRAND(); // dummy method call for field access
4    String p = Context.TELEPHONY_SERVICE;
5    Object o = this.getSystemService(p);
6    TelephonyManager tm = (TelephonyManager) o;
7    String countryCode = tm.getNetworkCountryIso();
8  + IfClass.ifMethod(countryCode, "RU"); // dummy method call for if statement
9    if(countryCode.equals("RU")){
10     performMaliciousActivity();
11   }
12 }
```

Listing 6.1: Example of app instrumentation performed by DIFUZER (Lines with "+" represent added lines).

We gathered a list of 618 unique methods for internal values.

External: In the case of external properties, a developer can get sensitive information from three channels: 1) SIM card, 2) Internet Connection, and 3) GPS chip. The process to collect the source methods is similar to the one followed with `Cursor` class, except we do not know in advance the name of the classes to inspect. Therefore we relied on a heuristic to identify such classes: for each SDK version, we listed all the classes and kept only those with class names containing the following words: "Sms, Telephony, Location, Gps, Internet, and Http". Once the classes were retrieved, we listed the methods for each class and kept those starting by "get" or "is". The intuition is the same as in the case of internal sources.

We gathered a list of 794 unique methods for external values. Finally, after combining sensitive sources from internal and external values, our list contains 1285 unique methods (127 duplicates).

Instrumentation

Performing taint tracking, as briefly described in Section 4, consists of a data flow algorithm that propagates the taint from a source method to a sink method.

Sinks related challenge: We remind that one objective of DIFUZER is to identify SHSOs' trigger entry points. Consequently, the taints that DIFUZER tracks are supposed to fall into *if statements*. However, being not a method call, an *if statement* cannot be considered as a sink when using state-of-the-art static taint analyzers [96, 5, 11]. A concrete example of what DIFUZER tracks is given in Listing 6.1. On line 7, *countryCode* variable is tainted from *getNetworkCountryIso()* source. This value is then used (line 9) to perform a test and trigger malicious activity (line 9). As an *if statement* is not considered a sink, a flow cannot be found.

Our approach overcomes this limitation by instrumenting apps. To accomplish this, the app code is first transformed into Jimple [45], the internal representation of Soot [24]. Then, DIFUZER iterates over every condition of the app, and for each condition, DIFUZER inserts a dummy method `ifMethod` with the variables involved in the condition as parameters. This `ifMethod()` is static and declared in a dummy class `IfClass` that contains all instrumented methods related to conditions. See line 8 in Listing 6.1.

Once the instrumentation is over, we dynamically register every newly generated method calls as sinks to FLOWDROID.

Sources related challenge: As described in Section 6.2.1, we consider, in this study, `Build` class' fields as sources. Since field accesses are not method calls, we follow the same process as for *if statements* to insert dummy methods. More specifically, DIFUZER

generates a static method call on-the-fly representing a field access from the `Build` class. Listing 6.1 depicts an example of this instrumentation process, where the dummy method `getBRAND()` of the dummy class `BuildClass` is inserted in line 3. Furthermore, newly generated method calls are registered as sources for taint tracking.

6.2.2 Anomaly detection

This section presents DIFUZER's second module, which relies on anomaly detection. In particular, we detail the unsupervised machine learning technique used to detect abnormal triggers.

Why a One-Class SVM?

A classical classification problem requires samples from positive and negative classes to build a model, which is then used to assign labels to test instances [97]. This induces possessing a reasonable amount of samples from two classes, which is not the case in our study. Indeed, the SHSO detection problem is challenging, and to the best of our knowledge, there is no ground truth made publicly available. Thus, using supervised learning in our study is not practical and present limited feasibility.

Therefore, we decided to rely on an unsupervised learning technique to detect SHSOs, particularly on a One-Class Support Vector Machine (OC-SVM) machine learning technique. An SVM algorithm was chosen due to its ability to generalize [98] and its resistance to over-fitting [99]. The general idea of OC-SVM is to identify the smallest hyper-sphere to include most of the samples of the positive samples [100]. A sample considered an outlier by the model means the data point is not in the hyper-sphere.

Features extraction

As already said, the second DIFUZER module's objective is to detect abnormal triggers with the intuition that these triggers are HSOs for which the likelihood of being a logic bomb is high, namely SHSOs. This module implements an OC-SVM algorithm that takes as input feature vectors computed from the triggers previously extracted from the entry points yielded by the first module of DIFUZER (cf. Figure 6.1).

To engineer anomaly detection features, we reviewed surveys [60, 101] and related-papers [102, 103, 15, 64] discussing Android malware and investigated the techniques used by malware writers to hide malicious code within apps. Eventually, we identified nine unique trigger/behavior features that are described in the following.

In the remainder of this section, we consider a trigger $\tau = (c, T_c, \Phi_c)$ and its guarded code $\Gamma = T_c \cup \Phi_c$ (cf. Chapter 4).

DIFUZER builds a feature vector $v =< S, N, D, R, B, P, M_1, S_1, J >$ for a given trigger where:

S: Number of sensitive methods used in guarded code. Intuitively, this feature represents how much a trigger controls the execution of sensitive methods. Indeed, while HSOs guard the execution of sensitive operations for performing sensitive activities [17], benign triggers, in the general case, perform benign activities, i.e., invoke few sensitive methods or not at all. To retrieve this value, DIFUZER iterates over every statement of Γ and recursively checks whether a sensitive method is called or not. For this purpose, we gathered a list of sensitive APIs constructed in previous work [51].

N: Is native code used in guarded code? Since analyzing native code is more challenging than Java bytecode [104], Android malware developers tend to hide malicious code from automated analyses in native code [102, 103]. Hence, this feature is a boolean value that, when set to 1, means native code is used in Γ, 0 otherwise.

D: Is dynamic loading used in guarded code? Dynamic class loading is not exclusively used in malware. However, as malware is becoming increasingly sophisticated, they use built-in capabilities like dynamic loading to hide from automated analyses [15]. Consequently, likewise native code, this feature is a boolean value set to 1 if dynamic loading is used in Γ, 0 otherwise.
R: Is reflection used in guarded code? Android malware writers tend to use more and more reflection-based code [15] since most of the state-of-the-art techniques overlook this property due to the challenging task of resolving it. Therefore, this feature is set to 1 if reflection is used in Γ, 0 otherwise.
B: Does guarded code trigger background tasks? Android apps rely on the Service component to run background tasks. Hence, with this feature, we aim at capturing the fact that the app under analysis performs stealthy operations without user knowledge. The intuition here is that SHSOs' role is to hide code both from security analysts and end-users (e.g., in the case of a logic bomb). This feature is set to 1 if background services are triggered in Γ, 0 otherwise.
P: Are parameters of condition used in guarded code? This feature captures the dependency of a condition to its guarded code. The hypothesis is that, in the case of SHSOs, the guarded code does not use values used in the condition since they represent different behaviors. To achieve this, DIFUZER performs a def-use analysis of the guarded code to verify if any variable used in the condition is used before being assigned a new value. If this is the case, the feature is set to 1, 0 otherwise.
M_1: Number of app methods called only in guarded code. With this attribute, we attempt to uncover the number of methods defined in the app called only in the guarded code of a trigger. The rationale is that app methods that are only used under a specific circumstance are likely to be defined only for this specific circumstance, representing hidden behavior [17]. To retrieve this number, DIFUZER queries the call graph (built using SPARK [50] algorithm) for each method call in the guarded code to verify if it has only one incoming edge (i.e., it is only called within the current method).
S_1: Number of sensitive methods called only in guarded code. In the same way as M_1, we aim to capture the number of sensitive methods only used in the guarded code of a given trigger.
J: Behavior difference between branches. Intuitively, two branches of an SHSO should be noticeably different. Indeed, of the two branches, one is considered the normal behavior (no or few sensitive operations) if the condition is not satisfied and the other as the sensitive behavior (sensitive operations) if the condition is satisfied [64]. Therefore, to compute this difference, DIFUZER first inter-procedurally retrieves sensitive method calls in both branches of a given trigger. Let X_{T_c} and X_{Φ_c} respectively be the sets of sensitive methods in the true and the false branch of a trigger. Therefore, to compute this difference of the two branches, DIFUZER relies on the Jaccard distance: $D_j(X_{T_c}, X_{\Phi_c}) = 1 - \frac{|X_{T_c} \cap X_{\Phi_c}|}{|X_{T_c} \cup X_{\Phi_c}|}$, which characterizes the behavior difference of the two branches. A value close to 1 means that both branches are dissimilar.

Training phase

To train our OC-SVM model, we need samples of a positive set, i.e., triggers considered normal. Therefore, we randomly chose 10 000 goodware (i.e., VirusToal [55] score = 0) from ANDROZOO [46]. Then, for each of these apps, we applied DIFUZER to extract a feature vector for each app's condition.

Afterward, we randomly chose 10 000 feature vectors[1] from those yielded by DIFUZER,

[1]The number of extracted vectors is orders of magnitude higher. However, for efficiency, we validated that a random set of 10 000 vectors yields an acceptable performance.

which we labeled as positive (i.e., part of the normal behavior). We then trained our One-Class Classification-based anomaly detector, leveraging LibSVM [105]. To ensure that the selected training set does not bias the trained model's performance, we split it and compute Accuracy in 10-fold cross-validation. Overall, we achieve a stable Accuracy of 99.91% on average.

6.3 Evaluation

To evaluate DIFUZER, we address the following research questions:
RQ1: What is the performance of DIFUZER for detecting Suspicious Hidden Sensitive Operations (SHSOs) in Android apps?
RQ2: Can DIFUZER be used to detect logic bombs? We address this question by considering three sub-questions:

- **RQ2.a:** Are SHSOs detected by DIFUZER likely logic bombs?
- **RQ2.b:** How does DIFUZER compare against TRIGGERSCOPE, a state-of-the-art static logic bomb detector?
- **RQ2.c:** From a qualitative point of view, does DIFUZER lead to the detection of non-trivial triggers/logic bombs?

RQ3: Can SHSO detection in goodware reveal suspicious behavior?

6.3.1 RQ1: Suspicious Hidden Sensitive Operations in the wild

In this section, we assess the efficiency of DIFUZER to find SHSOs on a dataset of malicious applications.
Dataset. To the best of our knowledge, no SHSO ground truth is available in the literature. Consequently, in this study, we considered 10 000 malicious Android apps as our malicious dataset. These apps were released in 2020, collected from the ANDROZOO [46] repository, and have been flagged as malware by at least five antivirus scanners in VirusTotal.

We contacted the authors of state-of-the-art approaches (e.g., HSOMINER [64], and TRIGGERSCOPE [17]) to get their artifacts (datasets and tools) for comparative assessment. Unfortunately, no artifact was made available to us.
Libraries. It has been shown in the literature [56, 106] that library code can affect analyses performed over Android apps since it often accounts for a larger part than the app's core code. Consequently, in this study, we considered two cases: (1) with-lib analysis (i.e., we consider the entire app code including library code); (2) without-lib analysis (i.e., we consider only developer code). We rely on the state-of-the-art list available in [56] to rule out libraries.
Post-Filter. As a precaution, before analyzing the results without libs, we listed the classes in which DIFUZER found potential sensitive triggers to search for redundant classes that could indicate libraries. We were able to filter out 19 additional libraries that were not listed in the list we used and provided by [56].

In the following, when referring to the analysis without libraries, we consider the 19 libraries previously presented as well as the libraries of the list in [56] as filtered. It accounts for a total of 5982 library classes and packages filtered.

Efficiency of Detecting SHSOs

We recall that DIFUZER is targeted at detecting SHSOs. While in RQ2 we investigate the likelihood for these SHSOs to be logic bombs, we first investigate the efficiency (with

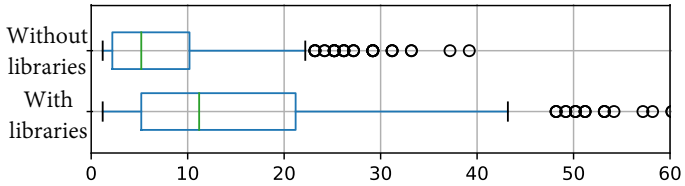


Figure 6.2: Distribution of the number of SHSO(s) per app in analyses with and without libraries (only apps with at least one SHSO are considered).

	Analysis with libs	Analysis without libs
Number of apps with SHSO(s)	339	259
Number of SHSOs	5575	2435
Number of SHSOs/app	16.4	8.2
Average # triggers (i.e., before Anomaly detection)	17.43	14.60
Average # SHSOs (i.e., after Anomaly detection)	0.56	0.24
Mean analysis time	35.63 s	33.54 s

Table 6.2: Results of the experiments executed on 10 000 malware with and without taking into account libraries.

RQ1) of Difuzer in the detection of SHSOs. We further perform an ablation study to highlight the performance of the anomaly detection module.

In Table 6.2, we report the results of applying Difuzer (with the anomaly detection step activated) on our 10 000 malware dataset. When analyzing the entire apps, Difuzer detects at least one SHSO in 339 apps (3.39%). Overall, Difuzer detects 5575 SHSOs in these 339 apps leading to an average number of 16.4 SHSOs per app. In comparison, when only the app developers' code is considered, Difuzer detects at least one SHSO in 259 apps (2.59%), with a total number of 2435 SHSOs detected and an average number of 8.2 SHSOs per app. We note that the 3140 (5575-2435) SHSOs that are not in the app developer code, are actually detected in 68 libraries suggesting that only a few libraries contain SHSOs. Figure 6.2 further details the distribution of detected SHSOs per apps.

These first results show that SHSOs indeed exist in malicious apps, but in relatively low number (in around 3% of the apps). However, when SHSOs are present in an app, they are not rare (on average, about 8 SHSOs per app in the developer code). Finally, SHSOs are more prevalent in library code than in app developer code, but only a few libraries contain SHSOs.

Table 6.2 also reports the average numbers of triggers before and after applying the anomaly detection step (i.e., the second module of Difuzer). Interestingly, we can see that this anomaly detection drastically reduces the number of triggers that are considered as SHSOs. Indeed, when considering the 10 000 apps, there are on average $174336/10000 \approx 17.43$ and $146018/10000 \approx 14.60$ triggers per apps (with or without libraries respectively) generated by the first module of Difuzer, i.e., by the taint analysis step. After the anomaly detection step, these numbers drop to $5575/10000 \approx 0.56$ and $2435/10000 \approx 0.24$ respectively, corresponding to a decrease of 96% and 98% respectively.

These results show that the anomaly detection step significantly impacts the number of detected SHSOs by significantly reducing the search space of triggers by up to 98%. This search space reduction is key when the ultimate goal is to detect malicious code and to support security analysts manual inspection (cf. Section 6.3.2).

We further inspect the SHSOs detected by Difuzer by focusing only on the app developer code (we do not consider library code). Table 6.3 lists the top 10 types of trigger that Difuzer was able to discover. The second column gives some examples of methods considered sources for the taint tracking to uncover SHSO entry points. We note the diversity of types of triggers that developers use. For instance, a developer can decide to

Trigger Type	Examples of methods	# T.	Trigger Type	Examples of methods	# T.
Database	getString, getInt, getCount	785	Location	getLastKnownLocation, getLongitude	84
Internet	getResponseCode, getResponseMessage	715	Wi-Fi	isWifiEnabled, getConnectionInfo	76
Build	getMODEL, getMANUFACTURER	374	Power	isScreenOn, isInteractive	47
Telephony	getDeviceId, getNetworkOperatorName	97	Audio	getStreamVolume, isMusicActive	37
Connectivity	getActiveNetworkInfo, getNetworkInfo	88	Camera	getCameraIdList	28

Table 6.3: Top ten trigger types DIFUZER discovered in the developer code. (T. = Triggers)

trigger (or not) the sensitive code if: (Database trigger type) specific values are present in databases (e.g., contacts, messages); (Internet trigger type) external orders say so; (Build, Telephony, and Camera trigger types) the device is not an emulator; (Connectivity, and Wi-Fi trigger types) the device has Internet access; (Location rigger type) the user is in a pre-defined location; Note that the methods in Row 3 have been dynamically generated by DIFUZER during instrumentation to track the Build class's field values.

Regarding the component types in which DIFUZER found SHSOs, 90% of SHSOs are in methods of "normal" classes, i.e., not Android components. SHSOs are found in `Activities` in 9% of the cases. However, they are rarely found in `Services` and `Broadcast Receivers` (less than 1%).

Manual Analyses

Since static analysis approaches often suffer from false alarm issues, i.e., they report a large proportion of false positive results, we decided to verify the detection capabilities of DIFUZER manually. To that end, we randomly selected a statistically significant sample of 102 apps out of the 259 apps in which SHSOs exist in developer code, with a confidence level of 99% and a confidence interval of ± 10%. Only one sample was found to be a false positive result. Indeed this app verifies if it is running in an emulator by comparing `Build.PRODUCT`, `Build.MODEL`, `Build.MANUFACTURER`, and `Build.HARDWARE` against well-known strings such as "generic", "Emulator", "google_sdk", etc. This test seems sensitive, but the guarded code displays the following message to the user: "Scooper Warning: App is running on emulator.". Therefore, DIFUZER achieves a precision of 99.02 % to find *Suspicious Hidden Sensitive Operations* on this dataset. We release the annotated list of 102 apps that were manually checked for transparency in the project's repository.

Analysis Time

The last row in Table 6.2 reports DIFUZER analysis time. DIFUZER outperforms state-of-the-art trigger detectors with an average of 33.54 s per app (35.63 s for the analysis with libraries, with an average DEX size of 7.03 MB per app), making DIFUZER suitable for large-scale analyses. In comparison, state-of-the-art tools such as TRIGGERSCOPE [17] and HSOMINER [64]) require 219.21 s and 765.3 s per app respectively. Note that 85.42% (i.e., 28.65 seconds on average) of this time is reserved for the taint analysis. Also, 24 apps (0.24%) reached the timeout (i.e., 1 hour) before the end of the analysis.

RQ1 answer: DIFUZER detects SHSOs in Android malware with high precision, i.e., 99.02 % in less than 35 seconds on average. Among the average 14.6 HSOs identified in an app based on triggers spotted by static taint analysis, only 2% are suspicious according to anomaly detection, which shows that DIFUZER is effective in reducing the search space for manual analysis.

6.3.2 RQ2: Can Difuzer detect logic bombs?

In this section, we ① evaluate DIFUZER's efficiency in detecting logic bombs (RQ2.a), ② compare it against TRIGGERSCOPE (RQ2.b), and ③ discuss logic bomb use cases in real-world apps (RQ2.c).

RQ2.a: Are SHSOs detected likely to be logic bombs?

Until now, we have shown that DIFUZER is effective in detecting SHSOs. From a security perspective, however, we must further show that these SHSOs are actually malicious. In other words, are these SHSOs likely to be logic bombs? Unfortunately, such an assessment is challenged by the lack of ground truth in the literature. We, therefore, require extra manual analysis effort of reported results.

Initial Manual Analysis: In the previous Section 6.3.1, we present our manual analysis of SHSOs detected in 102 apps. We further checked whether the detected SHSOs contained malicious code during this analysis. In particular, for each app under analysis, we gathered information about the reason it was flagged by antiviruses (e.g., on VirusTotal). Then, in the guarded code of the potential SHSO found by DIFUZER, we looked for malicious behavior matching our information previously gathered. For instance, if: (1) an app is labeled as being a trojan stealing the device's information; (2) the potential SHSO is performing emulator detection (e.g., calling `System.exit()` method if the device is running in an emulator); and (3) the behavior exhibited in the code guarded by the condition detected by DIFUZER is gathering the device's information (e.g., unique identifier, current location, etc.) and sending it outside the device, the SHSO is considered a logic bomb.

Eventually, 30 apps (i.e., 29.7%) were manually confirmed to be logic bombs, i.e., the SHSOs were triggering malicious code.

Semi-Automated further Analysis: Manual investigation is time-consuming. This is the reason why we inspected 102 apps, and not all 259 apps reported to having at least one SHSOs within the developer code parts. To quickly enlarge the set of identified logic bombs, we decided to follow a simple but efficient process. It is known that malicious developers often reuse the same piece of code in different apps [101]. Therefore, for each already identified logic bomb, we search for similarities (i.e., SHSOs found in the same class name, method name, and type of trigger used) in SHSOs contained in the 157 $(259 - 102)$ remaining apps. Our analysis yielded 16 additional apps containing logic bombs that were manually verified and confirmed. Eventually, our manual analysis yielded 46 Android apps.

Discussion about HSO, SHSO and Logic Bombs: In the literature [64, 17], HSO is consistently defined as a sensitive operation that is hidden by specific triggering conditions. Nevertheless, the notion of "sensitive operation" is not clearly delineated, which challenges comparison across approaches. In our work, we postulate that detecting HSOs is an important first step, but it is not enough to help security analysts. Indeed, as shown by our manual analysis, many HSOs are sensitive but not necessarily suspicious. As a result, most of the detected HSOs are legitimate and do not require any inspection effort from security analysts.

In this context, if the goal is to detect real security issues and reduce the burden of security analysts, a tool such as HSOMINER [64] which detects *HSOs* in 18.7% of apps within a set of over 300 000 apps (including malicious and benign apps) appears to be unpractical. In contrast, DIFUZER detects *suspicious HSOs* in 3.39% of the analyzed apps (when libraries are considered), and our manual analyses confirm that in about 30% of the apps, these SHSOs are logic bombs, making the work of security analysts easier. Though both HSOMINER dataset and our dataset are different (we were not able to get the HSOMINER's authors dataset), if we compare the 18.7% of apps with HSOs reported

by HsoMiner, with the 3.39% reported by Difuzer, we can say that Difuzer reduces the search space by up to 81.9% ($(18.7-3.39)\times\frac{100}{18.7}=81.9$) to accelerate the identification of logic bombs.

RQ2.a answer: By triaging HSOs to focus on suspicious ones based on anomaly detection, Difuzer was able to reveal 30 logic bomb instances in a sampled subset of malware apps having SHSOs.

RQ2.b: How does Difuzer compare against TriggerScope, a state-of-the-art logic bomb detector?

In the absence of a public ground truth for Android logic bomb instances, we perform experimental comparisons against the TriggerScope state-of-the-art detector in the literature that relies on static analysis. Although TriggerScope is not publicly available, we are able to build on a replication based on technical details provided in the TriggerScope's paper [17].

Overall, our approach differs from TriggerScope's by three major differences: ① **Technique**: TriggerScope uses symbolic execution to tag variables with a limited number of values, we use static data flow analysis; ② **Target**: TriggerScope detects hidden sensitive operations (i.e., whether at least one sensitive method is called within the guarded code of a trigger), whereas Difuzer 's goal is to detect suspicious hidden sensitive operations (i.e., the guarded code is sensitive and implements an abnormal behavior); and ③ **Approach**: TriggerScope maintains a list of sensitive methods and uses the occurrence of any of them as the sole criterion, Difuzer implements an anomaly detection scheme where the presence of sensitive methods is one feature among many others. While TriggerScope and Difuzer both rely on list of sources to find triggers of interest, TriggerScope handpicks a limited set of methods, whereas Difuzer 's list is based on a systematic mapping (cf. Section 6.2.1 - we leverage patterns to systematically search for sources).

Does* TriggerScope *identify as logic bombs the SHSOs flagged by* Difuzer*?

We applied TriggerScope on the subset of 102 apps where Difuzer identified an SHSO (cf. Section 6.3.2). The objective is to check whether TriggerScope is more or less accurate than Difuzer. Typically, among the 30 logic bombs that have been manually verified as true positives, how many are detected by TriggerScope. Similarly, does TriggerScope detect logic bombs (manually verified as true positives) that Difuzer could not? Figure 6.3 illustrates the differences in logic bomb detection (left figure). Overall:

- TriggerScope did not flag any logic bomb that Difuzer did not.
- TriggerScope could only detect 2 logic bombs among the 30 logic bombs that Difuzer correctly identified.
- As reported in the literature [37], TriggerScope exhibits a very high false positive rate at 94.6%: 35 among its 37 detections are false positives (the rate for Difuzer is 70.6%, 72/102).

Does* Difuzer *fail to flag as SHSOs the logic bombs detected by* TriggerScope*?

We recall that, contrary to Difuzer, which builds on anomaly detection, TriggerScope is restricted to detect only logic bombs where the trigger involves location-, time-, and SMS-related properties. Aligning with the assessment of Difuzer, we applied TriggerScope on our set of 10 000 malware. TriggerScope reported 591 logic bombs in 149 apps (∼4/app): 98.6% of the reported cases are time-related. In the absence of ground

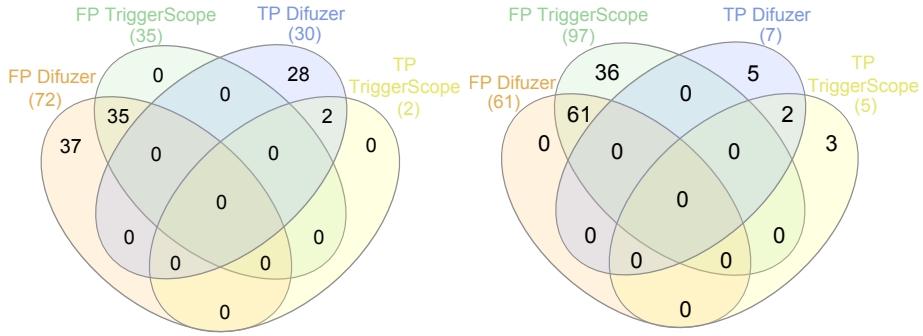


Figure 6.3: Venn Diagram representing results of TriggerScope and Difuzer on 102 apps originally detected by Difuzer on the left, and TriggerScope on the right. (FP = False Positive, TP = True Positive)

truth, we again propose to manually verify a random sample set of reported logic bombs. To facilitate comparison with Difuzer, we sample 102 apps (we simply considered the same number of apps as in the previous question), and manually confirmed that for 97 (95.1%) apps, the reported logic bombs are false positives. In 5 (4.9%) apps, we found at least one reported logic bomb to be a true positive.

We further check whether on these 102 apps where TriggerScope reported a logic bomb, Difuzer also flags any case of SHSO: Difuzer flagged 68 apps as containing SHSOs, among which 7 are manually confirmed to be logic bombs. The details of the comparison between TriggerScope and Difuzer are presented in the Venn Diagram in Figure 6.3 (right figure). We note that:

- 2 logic bombs are detected by both Difuzer and TriggerScope.
- 5 SHSOs detected by Difuzer are actual logic bombs but not detected by TriggerScope. Indeed, TriggerScope is limited by its focus on time, location, and SMS-related triggers.
- 3 logic bombs are detected by TriggerScope, but not detected by Difuzer. Our prototype implementation considers a limited list of sources, which do not cover those 3 logic bomb cases.

Although we do not have a complete ground truth (with information about all cases of logic bombs), confirming and comparing detection reports by Difuzer and TriggerScope offers an alternative to assess to what extent each may be missing some logic bombs. The results described above suggest that Difuzer suffers significantly less from false negative results than TriggerScope.

RQ2.b answer: Overall, Difuzer outperforms TriggerScope by detecting more logic bombs more accurately (wrt. false positives), and by missing less logic bombs (wrt. false negatives).

RQ2.c: From a qualitative point of view, does Difuzer lead to the detection of non-trivial triggers/logic bombs?

In this section, we discuss two real-world apps in which Difuzer revealed logic bombs that cannot be detected by TriggerScope.

Database	Internet	Location	Connectivity	Audio	Telephony	Wi-Fi	View	Activity	Build
897	283	264	74	63	58	25	21	19	19

Table 6.4: Top ten trigger types used by benign Android apps.

Advertisement Triggering. DIFUZER revealed an interesting logic bomb in "com.walkthrough.knife.assassin.hunter.baoer" app, which is an adware app of the HiddenAd family. The app uses the `android.app.job.JobService` class of the Android framework to schedule the execution of jobs (the developer can handle the code of the job in `onStartJob` method). In the `onStartJob` method, the app takes advantage of the `PowerManager` of the Android framework to check if the device is in an interactive state (i.e., the user is probably using the device) with method `isScreenOn()`. If this is the case, the app displays advertisements to the user and schedules the same class's execution after a certain time.

Data Stealer. Logic bombs can also be used to trigger data theft under the condition that the data is available. For instance, in the app "com.magic.clmanager", which is a Trojan (hidden behind a cleaning app) capable of stealing data on the device, DIFUZER found a logic bomb related to the device's unique identifier. Indeed, in method `d(Context c)` of the class `c.gdf`, a check is performed against the value returned by method `getDeviceId()` to verify if the value matches specific values (emulator detection) in a given file named "invalid-imei.idx". In the case, the app considers that the device is not an emulator, it triggers the stealing of sensitive information about the device such as the current location, phone number, information on the camera, information about the Bluetooth, disk space left, whether the device is rooted or not, the current country, the brand, the model, information about the Wi-Fi, etc. Afterward, this information is written in a file and sent to a native method for further processing.

6.3.3 RQ3: SHSOs in benign apps

Until now, we have focused on malware. However, SHSOs are not exclusively found in malicious apps [64]. Therefore, in this section, we intend to conduct a study on benign applications.

Results. As confirmed in Section 6.3.1 and in previous studies [64, 17], benign libraries and benign Android apps implement HSOs. Our study confirms this finding. Even more, 354 benign apps (3.54%) were flagged by DIFUZER to contain suspicious HSOs. We further manually analyzed 20 apps randomly selected from our results and confirmed that they all contain at least one SHSO. Table 6.4 shows the different trigger types used in benign apps to trigger SHSOs. A significant result here is that benign apps use considerably less the "Build" trigger type (see Table 6.3 for comparison) than malicious apps. Similarly, the "Telephony" trigger type is less used in benign apps than in malicious apps. This induces that, in benign apps, decisions are less taken depending on values derived from methods like: `getDeviceId()`, `getNetworkOperatorName()`, `getPhoneType()`, `getMODEL()`, `getMANUFACTURER()`, or `getFINGERPRINT()`. A hypothesis would be that benign apps are less prone to recognize an emulator environment (and use this information to set triggering conditions).

Besides, we can see in Figure 6.4 that, in comparison with malicious apps, benign apps tend to have significantly fewer triggers per app.

Case Study

This section presents an SHSO of a benign app.

Benign App. The app we consider in this case study is "no.apps.dnbnor". DIFUZER

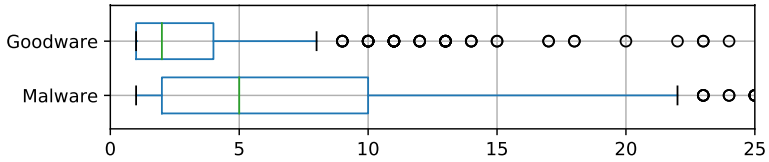


Figure 6.4: Distribution of the number of SHSO(s)/app in goodware and malware (apps with at least 1 SHSO are considered).

detected an SHSO in method `<bom.∮: java.lang.String ···()>` which tests if the value of `Build.CPU_ABI` or `Build.CPU_ABI2` is equal to pre-defined values stored in a file. In the case a match is found, it triggers the copy of a native code file into a second file. The native code file name is in the form: "lib/" + `str` + "/lib" + `f9` + ".so". The `str` variable represents a `CPU_ABI` value and the `f9` variable represents a string to designate the file. This file is then opened and eventually copied in the user data directory of the running app.

Although not malicious in this case, this behavior is suspicious, and DIFUZER was able to reveal it.

Malicious activities in Google Play

We now illustrate how DIFUZER contributed to removing 8 apps whose behavior was potentially harmful to users (in the form of aggressive, unsolicited, and intrusive ads) in Google Play. Developers of such *adware apps* managed to evade classical checks performed in Google Play.

During our manual analyses of benign apps, we stumbled upon an app with an SHSO flagged by DIFUZER. Our inspection of the code suggested that the SHSO is not a logic bomb per se since it does not trigger the malicious code. However, during this manual analysis, we noticed that the app was apparently mainly designed to display advertising content aggressively. We downloaded the sample and executed it in an emulator to confirm our hypothesis. First, we noticed poor app design, poor quality, and low content. Then in nearly every screen (i.e., `Activity` component), we received embedded ads and full-screen ads. This behavior is characteristic of adware apps. After verification, we found that the app was still in Google Play with a relatively high number of downloads (a few thousands) but with negative comments. In fact, the app pretended to provide users with a "walkthrough" version of an existing game to display a profusion of ads on each screen.

We then searched in our analyzed apps if DIFUZER detected similar SHSOs. Eventually, DIFUZER detected three apps with the exact same SHSO and the exact same service proposed to the user (walkthrough games). We tested these apps to confirm they were adware. They were also still in Google Play.

We then checked if similar "walkthrough games" were also still in Google Play and not in our initial dataset. Therefore, we searched for apps made by the same developers of the three previous apps detected by DIFUZER. We also searched for "walkthrough games" in Google Play and browsed the resulting apps. We inspected the newly collected apps and confirmed they were adware apps. Eventually, we identified 8 apps with the same adware behavior.

We contacted Google to report these 8 apps. They were removed in less than two weeks from Google Play. We make available the samples in the project's repository.

RQ3 answer: Our experiments show that SHSOs are present in benign apps and in widely-used libraries. We have seen through real-world examples that DIFUZER can reveal potentially harmful applications (PHA) and raise alarms concerning some apps' potential maliciousness. Overall, DIFUZER contributed to removing 8 adware apps from Google Play.

6.4 Limitations and Threats to Validity

An essential step in our approach is the identification of SHSOs entry points. To do so, DIFUZER relies on state-of-the-art tool FLOWDROID [5]. Therefore, it carries the analysis limitations of FLOWDROID, i.e., unsoundness regarding reflective calls [8], dynamic loading [107], multi-threading [108] and native calls [109].

Although our approach proved to be efficient to detect SHSOs and logic bombs, feature selection can impact the performances. Indeed, feature engineering is a challenging task and can be prone to unsatisfactory selection since it does not capture *everything*.

Besides, our approach is based on SHSO entry points detection using taint analysis, which relies on sources and sinks methods. Sinks are not an issue in our approach since they always represent *if conditions*. However, sources selection is at risk since they have been selected systematically, using heuristics and human intuitions. Therefore, our list of sources might not be complete.

Although, we have implemented TRIGGERSCOPE by strictly following the description in the original paper, our implementation might not be exempt from errors.

In the absence of a-priori ground truth, some of our assessment activities rely on manual analysis based on our own expertise. While we follow a consistent process (e.g., we carefully verify the hidden behaviour implementation against the antivirus report), our conclusions remain affected by human subjectivity. Nevertheless, we mitigate the threat to validity by sharing all our artefacts to the research community for further exploitation and verification.

6.5 Related work

Logic bombs in general. Hidden code triggered under specific conditions is a concern in many programming environments. The literature includes studies of the logic bomb phenomenon in programming prior to the Android era [110, 30] and targeting the Windows platform for example. Since then, various approaches have been proposed to tackle the challenging task of trigger-based behavior detection [111, 112, 113, 114, 115]. State-of-the-art techniques for the detection of trigger-based behaviour are varied and leverage fully-static analyses [44, 17, 70], dynamic analyses [14], hybrid analyses [30, 66], and machine-learning-based analyses [64].

Trigger-based behavior detection for Android DIFUZER combines static taint analysis and unsupervised machine learning techniques. Our closest related work is thus HSOMINER [64], which relies on static analysis and automatic classification to detect *HSOs*. Contrary to our work, however, HSOMINER is not targeting suspicious HSOs and therefore does not focus on logic bombs.

Fratantonio et al. [17] proposed TRIGGERSCOPE, an automated static-analysis tool that can detect logic bombs in Android apps. TRIGGERSCOPE leverages a symbolic execution engine to model specific values (i.e., SMS-, time-, location-related variables). TRIGGERSCOPE models conditions using *predicate recovery*. It combines symbolic execution results and path predicate recovery results to infer suspicious triggers. Finally, potential suspicious triggers undergo a control dependency step to verify if it guards sensitive operations. Nevertheless, the whole approach relies on static analysis to check defined

properties of suspiciousness. In contrast, DIFUZER takes advantage of unsupervised learning to discover abnormal (hence suspicious) trigger-based behavior.

Anomaly detection for security. We note that the idea of using anomaly detection to detect malware has been presented in the Avdiienko et al.'s paper [116]. Indeed, they present MUDFLOW that relies on anomaly detection to spot malware for which sensitive data flows deviate from benign data flows. It proved to be efficient by detecting more than 86% malware. While our approach is also based on anomaly detection to triage *abnormal* triggers (i.e., suspicious sensitive behavior) that deviate from normality (i.e., normal triggers/conditions), the end goal of both approaches is different. Indeed, MUDFLOW addresses a binary classification problem to discriminate malware from goodware. In contrast, DIFUZER addresses the problem of detecting and locating *Suspicious Hidden Sensitive Operations* that are likely to be logic bombs in Android apps.

Malicious behavior detection in Android apps. Malware detection does not only focus on trigger-based malicious behavior. Indeed, the Android security research community worked on tackling general security aspects [60, 117, 118, 119, 120]. In the literature, numerous approaches have been proposed to detect Android hostile activities. Among which, machine-learning techniques [72], deep-learning techniques [121], static analyses through semantic-based detection [122], privacy leaks detection [6, 5, 123], as well as dynamic analyses [71, 12, 16]. Each of these approaches tackles a particular aspect of Android security. Therefore, analysts could combine our approach with the aforementioned techniques to detect a wide variety of Android malicious behavior more efficiently.

6.6 Summary

We proposed DIFUZER, a novel approach for detecting *Suspicious Hidden Sensitive Operations* in Android apps. DIFUZER combines bytecode instrumentation, static interprocedural taint tracking, and anomaly detection for addressing the challenge of accurately spotting relevant SHSOs, which are likely logic bombs. After empirically showing that our prototype implementation can detect SHSOs with high precision (i.e., 99.02 %) in less than 35 seconds per app, we assessed its capabilities to reveal logic bombs and demonstrate that up to 30% of detected SHSOs were logic bombs. We therefore improve over the performance of the current state of the art, notably TRIGGERSCOPE, which yields significantly more false positives, while detecting less logic bombs. Finally, we apply DIFUZER on goodware to investigate potential SHSOs: DIFUZER eventually contributed to removing 8 new adware apps from Google Play.

Chapter 7

A Dataset of Android Applications Automatically Infected with Logic Bombs

In this chapter, we propose a dataset of Android apps automatically infected with logic bombs. Indeed, the research community has proposed various approaches and tools to detect logic bombs. Unfortunately, rigorous assessment and fair comparison of state-of-the-art techniques are impossible due to the lack of ground truth. Hence, we present TriggerZoo*, a new dataset of 406 Android apps containing logic bombs and benign trigger-based behavior. These apps are real-world apps from Google Play that have been automatically infected by our tool* AndroBomb*. The injected pieces of code implementing the logic bombs cover a large pallet of realistic logic bomb types that we have manually characterized from a set of real logic bombs. Researchers can exploit this dataset as ground truth to assess their approaches and provide comparisons against other tools.*

This chapter is based on our work published in the following research paper:

- **Jordan Samhi**, Tegawendé F. Bissyandé, and Jacques Klein. TriggerZoo: A Dataset of Android Applications Automatically Infected with Logic Bombs. *In Proceedings of the 19th International Conference on Mining Software Repositories, Data Showcase, (**MSR**)*. 2022, 10.1145/3524842.3528020 [124].

Contents

7.1 Overview

The Android operating system is the most used worldwide in mobile devices [38]. Hence, Android security and privacy have become one of the major concerns of researchers. Every year, several thousands of threats are identified by antivirus companies spanning a wide range of maliciousness (e.g., trojan, adware, spyware, ransomware, etc.). To cope with malicious code proliferation, researchers set up several approaches that rely on static analysis [2, 4, 17, 27], dynamic analysis [12, 71, 14], machine-learning based analysis [72, 73, 28], or hybrid approaches [29, 31, 30].

Nowadays, malicious developers build their codebase to avoid detection from analyzers [17, 7, 32, 37, 33]. A notable technique used to bypass dynamic analyses consists in employing *logic bombs* that allow the malicious code to be triggered only under specific circumstances (e.g., at a specific date). In recent years, researchers have therefore proposed various techniques to uncover logic bombs in Android applications [17, 7, 64]. However, a common challenge in advancing the state of the art is the lack of shared benchmarks for the assessment and fair comparison of literature approaches.

The research literature already proposed various datasets of Android apps to encourage reproducibility and comparison between different approaches. For instance, Allix et al. proposed Androzoo [46], a growing repository now including about 18 Million Android apps. Arzt et al. released DroidBench [5], a test suite to evaluate Android taint analysers. Nielebock et al. proposed AndroidCompass [125] as a dataset of Android compatibility checks. Recently, Wendland et al. [126] released AndroR2, a dataset of bug reports related to Android apps and Li et al. [127] released AndroCT, a large-scale dataset of runtime traces of benign and malicious Android apps. However, in the research directions related to logic bombs, the community faces a challenge to build a comprehensive dataset due to the known difficulties in detecting logic bombs. Indeed, even if an app is detected as malware, identifying a logic bomb in malware requires extensive manual inspection and strong expertise. Logic bombs are often simple *if statements* with "unusual" conditional expressions. Yet, it is far from being trivial to distinguish a "logic bomb condition" from a "legitimate and normal condition". The research community lacks an important artifact in the logic bomb detection domain, i.e., an Android app dataset that contains logic bombs with information about their localization in the apps.

In this work, we propose a new dataset of Android apps containing logic bombs and benign trigger-based behavior to the research community. This dataset, named TriggerZoo, contains 406 apps, from which 240 are infected with logic bombs, and 166 apps contain benign trigger-based behavior. It was generated by applying our dedicated tool, AndroBomb, on 2000 apps from Google Play. TriggerZoo is meant to facilitate research on logic bomb detection. Specifically, TriggerZoo will serve as a base for new approaches to detect logic bombs, assess new tools, and compare with other approaches. Besides, since AndroBomb has been developed with a modular approach, it is easy to add new trigger and behavior types.

The main contributions of our work are as follows:

- We propose TriggerZoo, a new reusable dataset of 406 Android apps infected with trigger-based behavior and their localization with 10 trigger types and 14 behavior types.
- We also propose AndroBomb, an extensible framework to inject trigger-based behaviors into Android apps automatically.
- We provide performance results of two state-of-the-art works Difuzer and TriggerScope on the TriggerZoo dataset.

TriggerZoo apps are made available in the AndroZoo repository, where they are responsibly shared with authenticated researchers only. TriggerZoo apps' hashes, and labels are available at:

https://github.com/JordanSamhi/TriggerZoo[1]

In the same way, and to avoid encouraging malware development, AndroBomb is only available to authenticated researchers. AndroBomb's instructions and appropriate files are available at:

https://github.com/JordanSamhi/AndroBomb[2]

7.2 Dataset Construction Methodology

In this section, we first describe AndroBomb used to construct TriggerZoo. Secondly, we give details about the process we followed to generate our dataset and describe it.

7.2.1 AndroBomb: Automatically Infecting Android Apps

AndroBomb has been designed to inject a trigger-based behavior (malicious or benign) in a specific location in an Android app. This trigger-based behavior is characterized by a *trigger type* (e.g., time check) and a *guarded code type* (e.g., the stealing of private information). In Figure 7.1, we present an overview of the AndroBomb approach. AndroBomb is made of three main parts: ① a mechanism for pinpointing an insertion point based on call graph construction and control flow analysis that serves to identify a method in which a trigger-based behavior can be inserted; ② an infection step where a trigger-based behavior is generated given a *condition type* and a *guarded code type*, and inserted in the insertion point; and ③ a repackaging step where the APK is updated with new permissions (if required), new native code files (if required), aligned, and signed to generate an infected APK file.

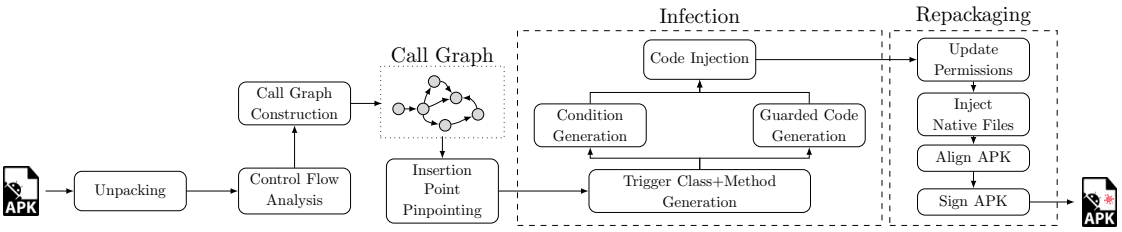


Figure 7.1: Overview of the AndroBomb approach to infect an Android app.

Insertion point Pinpointing Mechanism

We intend to inject trigger-based behavior into Android apps. The idea is to inject these trigger-based behaviors in methods ① highly likely to be executed at runtime, ② present in the developer code. To find these methods, AndroBomb relies on Flowdroid [5] and Soot [24] which provide a control flow analysis that is used to generate a call graph. To generate a call graph, we set Flowdroid to use the SPARK algorithm [50]. Then, AndroBomb builds a set of methods M that contains all the methods in the APK that are declared in a class for which the fully qualified name starts with the app's package name, i.e., it is a developer class. Indeed, we want to inject the logic bomb into the developer code to simulate malicious intent from the developer. M is then filtered to produce a new set of methods M_{cg} that only retains methods that are present in the call graph previously

[1]DOI: 10.5281/zenodo.5907916, Access: https://doi.org/10.5281/zenodo.5907916
[2]DOI: 10.5281/zenodo.5907924, Access: https://doi.org/10.5281/zenodo.5907924

generated, i.e., they may be called during execution. Eventually, AndroBomb randomly chooses a method in M_{cg} that will act as the *insertion point*.

Infection

To infect an app, AndroBomb needs two pieces of information: ① the trigger type used to activate the trigger-based behavior; and ② the guarded code type. This information is not static and is given as options to AndroBomb. As already mentioned, AndroBomb relies on Flowdroid, hence AndroBomb manipulates Jimple code [45] which is the language used to perform code instrumentation [26] and code injection. After dynamically generating the class and the method in which the code generated will lie, AndroBomb generates the trigger (according to the given type) and the guarded code (according to the given type). These pieces of code are merged to constitute a single entity (i.e., condition + code triggered) and injected into the insertion point. Note that *trigger types* and *guarded code types* have been chosen from existing logic bombs found in previous research and reverse-engineering, as well as benign trigger-based behavior found in the same way.

Packaging

After infecting the app's code, one has to take care of any collateral effect. Therefore, AndroBomb adds any permission needed for the code injected [51]. For instance, if AndroBomb injects a piece of code that steals the current location of the device and sends it over HTTP, AndroBomb also injects the following permissions to the AndroidManifest.xml file:

- android.permission.ACCESS_COARSE_LOCATION
- android.permission.ACCESS_FINE_LOCATION
- android.permission.INTERNET

This is to ensure that no error occurs at execution time.

Besides, AndroBomb can inject pieces of code that might invoke native code [128]. However, to invoke a native function, the APK should contain the related .so file (i.e., native libraries). Thus, AndroBomb also injects the adequate .so files needed to invoke a native library. We have developed these libraries. Their source code is available in the project's repository.

Eventually, the resulting APK is aligned [129] and signed [130]. Therefore, it can be installed on any device or emulator.

As AndroBomb only injects pieces of code that do not change the state of the initial app, the overall behavior remains unchanged. The infected app can be dynamically analyzed and monitored using emulators or statically analyzed to search for *potential* logic bombs. In addition, AndroBomb has been developed in a modular manner, allowing the community to easily add new *trigger types* and new *guarded code types* to generate new apps for this dataset, which is by design prone to evolve. We believe that this work will serve the community and advance the logic bomb detection research field.

7.2.2 TriggerZoo

To generate TriggerZoo, we relied on AndroBomb which can, at the time of writing, handle the trigger and guarded code types described in Tables 7.1 and 7.2. Triggers and guarded code types have carefully been chosen from existing logic bombs and benign trigger-based behavior. In addition, the literature [17, 7, 37] describes several use cases for which the authors extracted the trigger and guarded code types. Even if we cannot guarantee covering all possible trigger and guarded code types, by relying on these state-of-the-art works, we are confident that TriggerZoo covers a large proportion of logic bomb types that could exist in the wild.

Trigger Types	
Type	**Description**
time	at a specific time or date
location	at a specific location
sms	if a specific sms is received
network	if Wi-Fi available or specific http response received
build	if specific Build.MODEL/PRODUCT/FINGERPRINT are set
camera	if the device possesses cameras
addition	a dummy test with a simple addition
music	if some music is active
is_screen_on	if device in interactive state
is_screen_off	if device not in interactive state

Table 7.1: Trigger types handled by AndroBomb to generate TriggerZoo

◯ = benign behavior, ⊛ = malicious behavior

Guarded Code Types		
Type	**Description**	
return	no behavior	◯
sms_imei	send the device IMEI number by SMS	⊛
stop_wifi	deactivate the device Wi-Fi connection	⊛
write_string	write a constant to a file in the device's memory	◯
write_phone_number	write the phone number to a file in the device's memory	⊛
set_text	set a constant to be displayed on the screen	◯
sms_string	send a constant by SMS	◯
http_location	sends the current location to remote server using HTTP	⊛
set_text_reflection	set a constant to be displayed on the screen using reflection	◯
exit	exits the app	⊛
native_log_string	log a constant using native code	◯
native_log_model	log the Build.MODEL information using native code	⊛
native_write_phone_number	writes the phone number to a file using native code	⊛
native_phone_number_network	sends the phone number to a remote server using native code	⊛

Table 7.2: Guarded Code types handled by AndroBomb to generate TriggerZoo

Dataset Construction

As our initial set, we randomly collected 2000 Google Play apps from the AndroZoo dataset. Then, for each app, we applied AndroBomb with the trigger and guarded coded types randomly generated among those available in Tables 7.1 and 7.2. Each app is instrumented to receive one single trigger-based behavior. The resulting dataset, namely TriggerZoo, comprises 406 Android apps infected with trigger-based behavior. There are several reasons why AndroBomb was not able to generate an infected app for all of the 2000 apps:

- No insertion point was found in the app due to our strong constraint: we only consider methods that are in classes for which the fully qualified name starts with the app package name. (28.9%).
- The infected APK could not be repackaged due to the limitations of third-party software. For instance, (1) Soot could not handle multi dex APKs for apps using an Android API level < 22; (2) The ManifestEditor library crashes due to buffer underflow (54.5%).
- AndroBomb crashes since, for some apps, the methods added during infection do not exist yet because they were added in a subsequent Android API level (16.6%).

However, these AndroBomb limitations are not critical since its final goal is not to be 100% operational for a specific task (e.g., malware detection [121] and GDPR compliance [131]) but to construct a valuable dataset for the community, which it was able to achieve.

Field	
Type	**Description**
sha256_original_app	The sha256 hash of the original app
class_infected	The class infected
component_type	The component type of the class infected
method_infected	The method infected
trigger_type	The trigger type used to infect the app
guarded_code_type	The guarded code type used to infect the app
depths	Depths of the insertion point method in the app call graph

Table 7.3: TriggerZoo fields

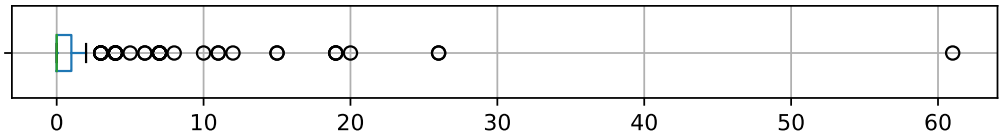


Figure 7.2: Infected methods' call graph depths in TriggerZoo

Dataset Description

TriggerZoo is composed of several files referenced in the project's repository:

- original_apps: SHA256 hashes of the 406 original apps.
- infected_apps: SHA256 hashes of the 406 infected apps.
- original_to_infected: links the original and infected apps.
- triggerzoo_labeled_dataset: the labeled dataset.

TriggerZoo is only available to authenticated researchers to have access to AndroZoo. Indeed, AndroZoo offers the authentication proxy for serving only the research community.

Format of the labeled dataset. The *triggerzoo_labeled_dataset* file in the project's repository describes in detail TriggerZoo with the fields available in Table 7.3. Each line of this file is composed of these 7 fields describing an app that has been infected.

Trigger and behavior types. TriggerZoo's repository shows two plots illustrating the number of apps infected with specific trigger and guarded code types. We can see that TriggerZoo covers a larger panel of trigger and guarded code types and their combination. Note that TriggerZoo covers 137 unique combinations of trigger and guarded code types. Besides, based on the guarded code types, TriggerZoo comprises 240 apps with malicious trigger-based behavior and 166 with benign trigger-based behavior.

Apps categories. TriggerZoo's project repository shows a third plot illustrating the different categories of TriggerZoo's apps. We were able to retrieve the category of 263 apps from the 406 in TriggerZoo. This is because the remaining 143 apps were not available on Google Play (i.e., they were removed from Google Play after being crawled by AndroZoo). We can see that TriggerZoo covers a large panel of categories, i.e., 34 unique categories.

Component types and Insertion Point Depth. We remind that TriggerZoo is built with AndroBomb that injects logic bombs or benign trigger-based behavior at random locations in the call graph of the app developer code. With this process, among our 406 apps, the trigger-based behavior has been inserted in methods of the `Activity` components for 381 apps and `Service` components for 25 apps. Figure 7.2 shows that for most apps, the call graph depth of the insertion point is low.

Dataset Validation

To ensure that the apps are infected by AndroBomb, we randomly selected a statically significant sample of 118 apps from the 406 supposedly infected apps, with a confidence level of 99% and a confidence interval of ± 10%, to manually inspect them. We confirm that 100% of the apps manually analyzed are infected, i.e., they contain the newly added code, the AndroidManifest.xml file is updated with new permissions (if needed in function of the APIs injected), and native files are indeed present in the app (if required in function of the APIs injected). Also, to ensure that the apps are not faulty and can be dynamically analyzed, we tried to install and run them on emulators. We confirm that 100% of the 118 apps manually analyzed can be installed and run without any problem. Hence, the instrumentation and repackaging processes do not impact the installation or the runtime processes.

7.3 Importance of TriggerZoo

Several state-of-the-art approaches have been proposed to detect logic bombs. TriggerScope was proposed in [17] as an automated tool to detect logic bombs. It relies on static analysis techniques such as symbolic execution, control flow analysis, predicate recovery, and control dependency. Recently, Samhi et al. [37] have released an open-source version of TriggerScope that they named TSOpen. Pan et al. [64] proposed HsoMiner, an approach relying on static analysis techniques such as control flow analysis, backward dependency graph, and trigger analysis. In addition, the authors proposed a machine-learning approach to automatically sort *hidden sensitive operations* (HSO). Recently, Samhi et al. [7] proposed Difuzer, an approach to triage logic bombs among *suspicious hidden sensitive operations* (SHSO). Difuzer relies on instrumentation techniques and taint tracking to identify SHSO entry points and triggers an anomaly detection engine to detect abnormal triggers.

The common points for these approaches are: ① they do not assess their tool on existing benchmarks. Hence, they cannot measure standard precision, recall, and f-1 score measures; ② they cannot compare their respective approaches against each other. To cope with these limitations, TriggerZoo can be used to measure existing and future tools' performances and will allow fair comparison.

In a first attempt to compare state-of-the-art approaches, and to assess TriggerZoo's usefulness, we executed Difuzer [7] and TSOpen [37] on the 406 apps present in TriggerZoo to search for logic bombs. Results are available in Table 7.4. We note that in the original Difuzer's publication, precision and recall metrics were provided based on a-posteriori manual checking, given the lack of benchmark. Thanks to TriggerZoo, comparisons such as the one presented in Table 7.4 are readily possible. Furthermore, note that the yielded results, with recall at 58%, suggest that TriggerZoo is "difficult" and will contribute to challenging future approaches. As a last remark, the a-posteriori comparison results presented in the Difuzer's paper [7] are confirmed on TriggerZoo. Indeed, as shown in Table 7.4, Difuzer clearly outperforms TriggerScope.

7.4 Summary

In this work, we presented two artifacts: ① TriggerZoo: a new evolving dataset of Android apps containing logic bombs and benign trigger-based behavior; ② AndroBomb: a new framework to infect Android apps with logic bombs and benign trigger-based behavior. TriggerZoo is meant to facilitate future logic bomb detector assessment and comparison. We also provide results of two state-of-the-art approaches, i.e., Difuzer and

	# apps	Difuzer		TSOpen	
		# analyzed	# flagged	# analyzed	# flagged
Malicious triggers	240	230	134	215	32
Benign triggers	166	156	41	148	15
Precision		76.6%		68.1%	
Recall		58.3%		14.9%	
F_1 score		66.2%		24.4%	

Table 7.4: Difuzer & TSOpen results on TriggerZoo

TriggerScope, on TriggerZoo, and confirm previous literature results. We believe that the research community will rely on this dataset to propose new approaches to detect logic bombs, thus improving Android apps' security and privacy. TriggerZoo is not frozen. It can evolve using AndroBomb to generate new samples with new logic bomb schemes. Both TriggerZoo and AndroBomb serve the community and support the logic bomb detection research direction.

Chapter 8

Part I Conclusion

In Part I, we presented our work to advance the state of the art regarding logic bomb detection in Android apps. The logic bomb detection problem is directly in line with our ambition to statically expose code that is unreachable from dynamic analyzers. Indeed, besides the fact that dynamic analyzers are "blind" for many code locations since the input provided might not satisfy the different constraints to reach them, the code guarded by logic bombs is *intentionally* made to bypass such analyzers. This enforces the motivation to use static analysis for identifying malicious behavior triggered under specific circumstances. Our work shows that, though this problem is challenging for dynamic and static analyzers, static analysis provides encouraging results toward pushing further the effort to make Android apps as free as possible from malicious behavior.

More specifically, we have made the following contributions: ① we have replicated an existing, though unavailable, approach to statically detect logic bombs in Android apps. Our study has brought to light the discrepancies between our observations and the original results. ② we have proposed a novel hybrid approach based on static analysis and anomaly detection to discriminate logic bombs from suspicious hidden sensitive operations in Android apps. ③ we have proposed a new dataset of Android apps automatically infected with logic bombs.

The logic bomb detection problem is *not solved yet*. Our work is a step toward the ambition to make Android apps free from malicious code. However, the research community still needs an important effort to reach this goal.

Part II

Statically Analyzing Code that is Unanalyzable for Existing Static Analyzers in Android Applications

Chapter 9

Motivation

Static analysis often relies on data flow analysis to compute sets of values at program points of interest (e.g., is a variable tainted? is a given variable positive? etc.). To perform a data flow analysis over a program, specific structures are constructed from this latter: ① control flow graphs that represent methods' statements and the paths that can be traversed during execution; and ② call graphs that represent the calling relationships between methods.

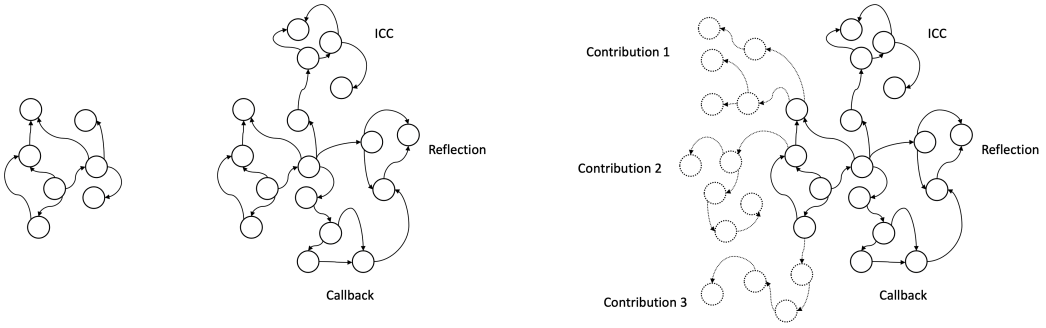
As we have seen in Chapter 2, in object-oriented programs such as Java, call graphs are, in part, over-approximated since there are mechanisms that prevent computing a 100% precise call graph.

To some extent, call graphs are also, in part, under-approximated when it is challenging to find proper targets for a method call. Indeed, these mechanisms, called *implicit calls*, prevent call graph construction algorithms used in static analyzers from automatically and natively finding the potential target of such calls. A concrete example is given in Listing 9.1 where a new `Thread` is created on line 4 and started on line 5. After the call to the `start()` method, the control flow will be delegated, inside the program, to the `run()` method on lines 11–13. Consequently, when construction the call graph, without human-defined heuristics, the given call graph would be incomplete, which would weaken any data flow analysis built on top of this call graph since the `run()` method would not be analyzed.

Android apps, which rely heavily on the Android framework and API methods, are mainly constructed over callbacks and implicit call mechanisms, which greatly challenge

```
1  public class MainActivity extends Activity {
2    @Override
3    protected void onCreate(Bundle b) {
4      myThread t = new myThread();
5      t.start();
6    }
7  }
8
9  public class myThread extends Thread {
10   @Override
11   public void run() {
12     // do something
13   }
14 }
```

Listing 9.1: An example of how implicit calls affect call graph construction



(a) Before SOTA (b) After SOTA (c) After our contributions

Figure 9.1: A simplified diagram of what would look like the call graph for a given Android app before the state-of-the-art existing contributions (a), after the state-of-the-art contributions (b), and after our contributions (c) presented in this manuscript. (SOTA = State Of The Art).

static analyses. Hopefully, implicit calls have already been explored in the literature. Many approaches have been proposed to improve call graph construction by connecting different types of implicit calls to potential target methods. Examples of these implicit calls are: life-cycle methods [5], reflection [8, 132], callbacks [133], or inter-component communication [6]. However, the current state of the art does not cover all types of possible implicit calls used in Android apps thanks to the Android framework.

Our work consist of extending this line of work to propose new techniques to make static analyzers handle and analyze more code. Figure 9.1a illustrates what a possible call graph could be for a given Android app before the current advances in the literature. Figure 9.1b shows what could be the call graph of the same Android app thanks to the state of the art advances in inter-component communication, reflection and callback modeling. Our ambition is to push further this body of work propose better approximation of what could be the *ideal* call graph of the same Android app. We illustrate this in Figure 9.1c where three of our contributions, presented in this part, contribute to improve Android apps' call graphs.

Chapter 10

Revealing Atypical Inter-Component Communication in Android Apps

In this chapter, we reveal that the Android framework provides different ways to enable inter-component communication (ICC) in Android apps, namely atypical inter-component communication. Hence, ICC models built by existing static analyzers are incomplete. To address this state-of-the-art limitation, we propose RAICC *a static approach for modeling new ICC links and thus boosting previous analysis tasks such as ICC vulnerability detection, privacy leaks detection, malware detection, etc. We have evaluated* RAICC*, demonstrating that it improves the precision and recall of state-of-the-art data leak detectors.*

This chapter is based on our work published in the following research paper:

- **Jordan Samhi**, Alexandre Bartel, Tegawendé F. Bissyandé, and Jacques Klein. RAICC: Revealing Atypical Inter-Component Communication in Android Apps. *In Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (**ICSE**)*, pages 1398-1409. IEEE, 2021, 10.1109/ICSE43902.2021.00126 [123].

Contents

10.1 Overview

Android apps heavily rely on the *Inter-component communication* (ICC) mechanism to implement a variety of interactions such as sharing data [134], triggering the switch between UI components or asynchronously controlling the execution of background tasks. Given its importance, the research community has taken a particular interest in ICC, reporting on various studies that show how ICC can be exploited in malicious scenarios: ICC can be leveraged to easily connect malicious payload to a benign app [101], leak private data [135, 11, 6], or perform app collusion [136]. These scenarios are generally executed by passing `Intent` objects, which carry the data and information about explicitly/implicitly targeted components [137]. Tracking information across Intents to link components that may be connected via ICC thus becomes an important challenge for the analysis of Android apps.

The resolution of ICC links (identification of the source and target components, type of the components, etc.) is a well-studied topic in the literature. Approaches such as EPICC [138], COAL/IC3 [139], SPARTA [140] or DroidRA [8] have contributed with analysis building blocks in this respect. The ICC links (also called ICC models) generated by these tools are key and even mandatory for several Android app analysis tasks. (1) In the case of data flow analysis, ICC poses an important challenge in the community: ICC indeed introduces a discontinuity in the flow of the analysis since there is no direct call to the target component life-cycle methods in the super-graph (aggregation of control flow graphs [25] of caller and callee methods in the absence of a single main method). Several tool-supported approaches such as Amandroid [11], IccTA [6] and DroidSafe [135] have been proposed in the literature to cope with this issue. To overcome the discontinuity in the analysis flow, all these three tools rely on an inferred ICC model to identify the target component and the ICC methods to artificially connect components. (2) In the case of Android malware detection, a tool such as ICCDetector [137] leverages the ICC model generated by EPICC to derive ICC-specific features that are used to produce a machine-learning model in order to detect a new type of Android malware. (3) In the case of vulnerability detection, EPICC leverages its own ICC model to detect ICC vulnerabilities, defined in [138] as the sending an Intent that may be intercepted by a malicious component, or when legitimate app components, –e.g., a component sending SMS messages– are activated via malicious *Intent* objects.

In all these cases, the proposed tools rely on comprehensive modeling of the ICC links. However, a major limitation in ICC resolution relates to the fact that state-of-the-art approaches consider only well-documented ICC methods such as `startActivity()`. Indeed, we have discovered that several methods from the Android framework can also be used to implement ICC, although the official Android documentation does not specifically discuss it [141, 142, 143]. Actually, ICC can also be performed by leveraging Android objects (e.g., `PendingIntent` or `IntentSender`) that have been little studied in the literature and through framework methods that can atypically be used to launch other components.

We have initially observed an atypical ICC implementation during the manual reverse engineering of an Android app that we identified as part of research on logic bomb detection. This app uses the method `set(int, long, PendingIntent)` of the `AlarmManager` class for triggering a `BroadcastReceiver` which in turn is used to launch a `Service` component. Such an implementation appeared suspicious since it seems artificially complex: it is possible to directly call the `sendBroadcast` method instead of leveraging an `AlarmManager`. We further performed extensive investigations and found that several dozens of methods of the Android framework can atypically start a component with objects of type `PendingIntent` and/or `IntentSender`. We use the term "atypical" to reflect the fact that, according to the method definitions, their role is not primarily to start a component (as ICC methods typically do) but to perform some action (e.g., set an alarm or send an

SMS). Unfortunately, with such possibilities, an attacker could rely on such methods to perform ICC-related malicious actions. Because they do not account for atypical methods in their models, existing state-of-the-art approaches would miss detecting such ICC links.

Our work explores the prevalence of *Atypical ICC* (AICC) methods in the Android framework as well as their usages in Android apps. We then propose an approach for resolving those AICC methods and an instrumentation-based framework to support state-of-the-art tools in their analysis of ICC.

In summary, we present the following contributions:

- We present findings of a large empirical study on the use of AICC methods in malicious and benign apps.
- We propose a tool-supported approach named RAICC for resolving AICC methods using code instrumentations in order to generate a new APK with standard ICC methods. We demonstrate that this instrumentation boosts state-of-the-art tools in various Android analysis tasks.
- We improve DroidBench [144] with 20 new apps using AICC methods for assessing data leak detection tools.

We make available our implementation of RAICC and the benchmark apps to reproduce our experimental results in the project's repository:

`https://github.com/JordanSamhi/RAICC`

10.2 How do state-of-the-art Analyzers handle ICC?

Android apps are composed of components that are bridged together through the ICC mechanism. The `Activity` component implements the UI visible to users while *Service* components run background tasks and *Content Provider* components expose shared databases. An app may also include a *Broadcast Receiver* component to be notified of system events. The *Manifest* file generally enumerates these components with the relevant permission requests.

Components are activated by calling relevant ICC methods provided in the Android framework. These ICC methods are also used to pass data through an *Intent* object, which may explicitly target a specific component or may implicitly refer to all components that have been declared (through *Intent Filters*) capable of performing the Intent actions.

The ICC mechanism challenges the static analysis of apps. Indeed, consider Listing 10.1 in which the `MainActivity` component launches the `TargetActivity` component. The discontinuity in the control flow is clear since there is no direct method call between `MainActivity` and `TargetActivity`. Off-the-shelf static analyzers that analyze normal method calls would not be able to detect the link between the ICC method `startActivity` and the `TargetActivity` component. Hence, if a data flow analysis is performed, none of the data flow values can be propagated correctly. This is since ICC methods trigger internal Android system mechanisms which redirect the call to the specified component.

Therefore, Android static analyzers must preprocess the application to add explicit method calls. That is what state-of-the-art tools like IccTA [6], DroidSafe [135] and Amandroid [11] do with different techniques. If we take the example of IccTA, it first relies on IC3 [139] to infer the ICC links. Among other information, IC3 identifies the ICC methods (e.g., `startActivity` in Listing 10.1) and resolves the target components (e.g., `TargetActivity` in Listing 10.1). Then, IccTA replaces any ICC method call with a direct method call that passes the correct `Intent`. Thus, the discontinuity disappears, and the link to the target component is directly available in the super-graph (see Figure 3

```
1  public class MainActivity extends Activity {
2    protected void onCreate(Bundle b) {
3      Intent i = new Intent(this, TargetActivity.class);
4      this.startActivity(i);
5    }
6  }
7
8  public class TargetActivity extends Activity {
9    protected void onCreate(Bundle b) {}
10 }
```

Listing 10.1: An example of how ICC is performed between two components.

of the IccTA paper [6]). The idea that we reuse in this work is the code instrumentation that allows preprocessing an app for constructing the missing links to be processed by any downstream analysis.

Nevertheless, in this work, we will see that state-of-the-art approaches only rely on well-documented methods for performing inter-component communication. We aim to improve their precision by revealing previously un-modeled ICC links.

10.3 Atypical ICC Methods

Static analysis of Android applications is challenging due to the specificity of the Android system's inter-component communication (ICC) mechanism. Therefore, as we have overviewed in Section 10.2, researchers have to come up with approaches for considering and resolving ICC. In this section, we show that one developer can perform atypical ICC by taking advantage of specific methods of the Android framework.

We define an *atypical ICC method* (AICC method) as a method allowing to perform an inter-component communication while it is not its primary purpose. These AICC methods rely on `PendingIntents` and `IntentSenders`. `PendingIntents` objects are wrappers for `Intents`. They can only be generated from existing `Intents` and describe those latter. They can be passed to different components and especially to different applications. When doing so, the receiving app is granted the right to perform the action described in the `PendingIntent` with the same permissions and identity of the source app. This introduces a security threat in which a component could perform an action for which it does not have the permission but it is granted this latter through the `PendingIntent`. This security threat has been studied by Groß et al. [145]. An important fact is that `PendingIntents` are maintained by the system and represent a copy of the original data used to create it. The `PendingIntents` can thus still be used if the original app is killed. IntentSenders objects are encapsulated into `PendingIntents`. They can be retrieved from a `PendingIntent` object via the method `getIntentSender()`. Basically, they can be used the same way than `PendingIntents` and represent the same artifact.

The abstract representation of AICC methods is shown in Figure 10.1. The upper part of the figure shows how standard ICC methods behave. They communicate with the Android system via `Intents` to execute another component. The lower part represents how AICC methods behave. They perform the action they are meant to do through the Android system and at the same time the `PendingIntent` or the `IntentSender` is registered in a token list in the Android system [146, 147]. The action may or may not influence the decision for the system to launch the component, depending on the AICC method. For example, a `PendingIntent` could only be launched in case of the success of the action. Also, the Android system can receive a cancellation of a token from the app. (e.g., cancel an alarm). In that case, the target component would not be launched.

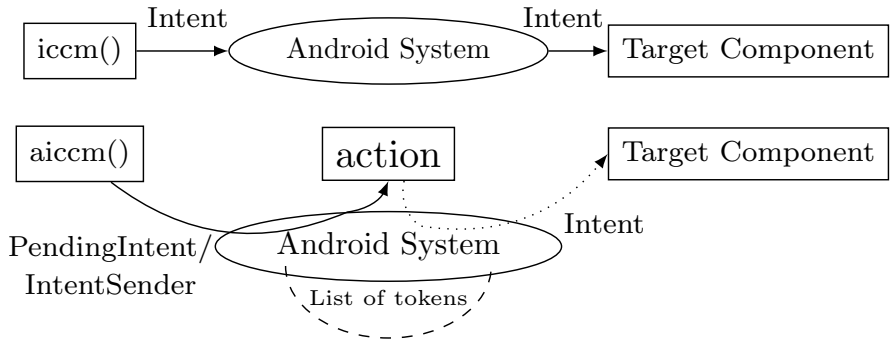


Figure 10.1: Difference between normal ICC method and AICC method. Tokens represent `PendingIntents` and `IntentSenders`. Action represents the primary purpose of the AICC method (e.g. send an SMS). An action might influence the list of tokens in the Android system, which will later process the list and send `Intents`. The dotted line indicates that the triggering of the target component may depend on the result of an action.

```
1  public void reconnectionAttempts() {
2    Calendar cal = Calendar.getInstance();
3    cal.add(12, this.elapsedTime);
4    Intent i = new Intent(this, AlarmListener.class);
5    intent.putExtra("alarm_message", "Wake up Dude !");
6    PendingIntent pi = PendingIntent.getBroadcast(this, 0, i, 134217728);
7    AlarmManager am = (AlarmManager) getSystemService("alarm");
8    am.set(0, cal.getTimeInMillis(), pi);
9  }
```

Listing 10.2: A simplified example of how the method `set` of the `AlarmManager` class is used in a malware.

The tokens represent the original data used for generating a `PendingIntent` or an `IntentSender`. It means that if the application modifies the `Intent` used to construct the `PendingIntent`, it does not affect the token as it is a copy of the original data. More importantly, if the application is killed, the list is maintained in the Android framework and the components can still be executed.

A concrete Example: As described in Section 10.1, while manually analyzing a malicious application, we noticed that it used the `AlarmManager` for performing ICC. The interesting piece of code of this malicious app is presented in Listing 10.2. We can see a `PendingIntent` created from an `Intent` targeting the component `AlarmListener`. The latter simply launches the `Service` component responsible for retrieving external commands via HTTP. For launching the class `AlarmListener`, the developer could have used the method `sendBroadcast` (`AlarmListener` extends `BroadcastReceiver`), but instead it used the AICC method `set(int, long, PendingIntent)`.

When we focus more on the way `PendingIntent` works, we understand why the developer used this technique. Indeed, in this example, the alarm is set up to go off after 5, 10, or 30 minutes. But what happens if the user closes the app before it goes off? In fact, the alarm will go off anyway and execute the target component. This is due to the fact that when setting an alarm, the `PendingIntent` is maintained by the Android system until it goes off or gets canceled. We can see the power of such a method to perform ICC. It could be used in different scenarios by an attacker to perform its malicious activities.

Furthermore, AICC methods carry information in `Intent` objects that are also embedded in `PendingIntent` or `IntentSender` objects. Therefore, they can carry different types of information, leading to potential sensitive data leaks. Our benchmark includes

examples scenarios for such leaks.

10.4 Approach

In this work, we aim at resolving those AICC methods through app instrumentation [26]. The goal for the new app is to be analyzable by state-of-the-art Android static analyzers. We first introduce in section 10.4.1 how we gather a comprehensive list of AICC methods. Then, in section 10.4.2 we describe how we leveraged this list of methods to improve the detection of inter-component communications leading to the increase of precision metrics of existing Android-specific static analyzers.

10.4.1 List of Atypical ICC Methods

As explained in previous sections, during the reverse-engineering of Android applications, we stumbled upon a malicious app making the use of the `set()` method of the `AlarmManager` class with a `PendingIntent` as a parameter to stealthily perform an ICC (in this case, to start a `BroadcastReceiver`). Thanks to this example, we realized that (1) `Intent` and method such as `startActivity` are not the only main starting points of ICC, (2) other objects (e.g. `PendingIntent`) and other methods (e.g. `AlarmManager.set()`) can play a similar role.

Motivated by this discovery, we were eager to check if this atypical mechanism is restricted to this `set()` method and this `PendingIntent` object. In other words, are there other atypical methods in the Android framework? Are there other classes such as `PendingIntent`? To answer these questions, we comprehensively analyzed the Android framework.

We retrieved from the Android framework, from SDK version 3 to 29 (versions 1 and 2 being unavailable), all the methods that take as a parameter an object of type `PendingIntent`. We obtained a list of 163 unique methods. The next step was to manually analyze all of them in order to only keep those allowing to perform ICC. The list reduced to 85 methods, indeed some methods have a `PendingIntent` as a parameter but cannot perform ICC (e.g., `android.bluetooth.le.BluetoothLeScanner.stopScan(PendingIntent)`).

We followed a simple heuristic to identify classes similar to `PendingIntent`. We search for all class names containing the string *Intent*. This search yielded 19 classes that we manually checked. Finally, we identified one new class, `IntentSender`, which, according to the Android documentation, has the same purpose as `PendingIntent`. We scanned the Android framework again to retrieve all the methods that take as a parameter an object of type `IntentSender`, and we discovered 17 new methods for performing atypical inter-component communication.

To improve confidence in our list of AICC methods, we performed further analyses. In particular, we downloaded the source code of Android and studied the implementation of some of the AICC methods we gathered. This approach aimed at finding patterns that we used to find similar usage in the Android framework. We assumed that other AICC methods use the same patterns. We also made some assumptions, e.g., considering the subclasses of those studied. Unfortunately, we were not able to uncover additional AICC methods.

Our list reached a length of 102 (85 + 17) methods at this stage. It was all without counting the 9 methods of `PendingIntent` and `IntentSender` classes that directly allow launching a component. For example, the `send()` method of the `PendingIntent` class allows to directly communicate with a targeted component, likewise for method `sendIntent()` of class `IntentSender`. Finally, our list reached 111 methods.

```
1  public class MainActivity extends Activity {
2    @Override
3    protected void onCreate(Bundle b)
4      [...]
5      Intent i = new Intent(this, TargetActivity.class);
6      String s = Context.TELEPHONY_SERVICE;
7      TelephonyManager tm = (TelephonyManager) getSystemService(s);
8      String imei = tm.getDeviceId();
9      i.putExtra("SensitiveData", imei);
10     PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);
11
12     (1.a) pi.send();
13
14     (2.a) IntentSender s = pi.getIntentSender();}
15     (2.b) s.sendIntent(this, 0, null, null, null);
16
17     (3.a) AlarmManager am = (AlarmManager) getSystemService("alarm");
18     (3.b) am.setExact(0, System.currentTimeMillis() - 100, pi);
19
20     (4.a) LocationManager l = (LocationManager) getSystemService("location");
21     (4.b) l.requestLocationUpdates(0, 0, new Criteria(), pi);
22     }
23 }
```

Listing 10.3: Examples of how AICC methods (in yellow) can be used to perform inter-component communication.

In Listing 10.3 we illustrate the usage of four AICC methods (chosen for their brevity). In the first lines (5–8), objects necessary to the AICC methods are instantiated. An `Intent` is instantiated at line 5. At lines 6–8, the device's unique identifier is retrieved and stored in the `imei` variable. In line 9, the IMEI is added as extra information in the intent. At line 10, the `PendingIntent` is instantiated with the intent containing the IMEI. Then, from line 12, we present four ways of launching the `TargetActivity` component through AICC methods.

We gathered a comprehensive list of 111 methods, called AICC methods, allowing us to perform atypical inter-component communication.

10.4.2 Tool Design

General Idea: The overview of our open-source tool called RAICC is depicted in Figure 10.2. The general idea is to instrument a given Android app to boost it by making it aware of ICC links. For instance, if a `PendingIntent` is used with an AICC method to start an activity, RAICC will instrument the app's source code by adding a method `startActivity()` with the right intent as parameter. This method is added at a point of interest in the app, i.e., just after the AICC method call. To perform this instrumentation, RAICC needs (1) to infer the possible values/targets of ICC objects (e.g., `Intent`); (2) resolve the type of the target component in order to instrument with the right standard ICC methods (e.g., `startActivity()` if the target component is an Activity, `startService()` if the target component is a Service, etc.).

Concrete Example: We illustrate the result of our approach with Listing 10.4. It shows the transformation that the Jimple code undergoes (shown as Java code for readability). The AICC method (program point of interest) appears on line 6. After inferring the target component type with the help of `COAL/IC3`, RAICC generates a new standard ICC method call right after the AICC method (i.e., at line 7) corresponding to

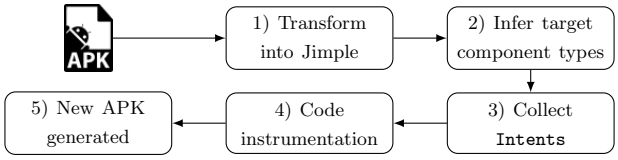


Figure 10.2: Overview of our open-source tool RAICC.

```
1    Intent  i = new Intent(this, TargetActivity.class);
2    PendingIntent p = PendingIntent.getActivity(this, 0, i, 0);
3    LocationManager l = (LocationManager)getSystemService("location");
4    Criteria c = new Criteria();
5    // program point of interest
6    l.requestSingleUpdate(l.getBestProvider(c,false), p);
7  + startActivity(i);
```

Listing 10.4: How RAICC would instrument an app. (Lines with "+" represent added lines)

this type, i.e., `startActivity()`. Indeed, the `PendingIntent` has been generated with the method `getActivity`, thus the target component type in the inferred values is defined as "a" in `COAL/IC3`, i.e., `Activity`. Also, RAICC is able to recover the `Intent` used to create the `PendingIntent` for using it as a parameter for the new standard ICC method call.

Details of each step involved in RAICC:

Step 1: The app is transformed into Jimple [45], the internal representation of the Soot framework [24] using Dexpler [49].

Step 2: RAICC leverages IC3 [139] which is able to infer all possible values of ICC objects using composite constant propagation at specific program points. To this end, we created model files using the COAL [139] declarative language to query each of the AICC methods during program analysis and retrieve the values of the parameters we need (i.e., `PendingIntent` and `IntentSender`).

Given that they are built from `Intent` objects, IC3 is able to identify all subparts which compose the objects (e.g., action, category, extras, URI, etc.). The most important artifact for our instrumentation is the types of potential target components. It is inferred by COAL given its specification, i.e., it is able to get the target component type by recognizing methods for creating `PendingIntents` (e.g., `getActivity`). Indeed, one can easily see the difference between a conventional ICC method and an AICC method: standard ICC methods explicitly describe the type of component that will be launched (e.g., `startActivity()` for an activity, `startService()` for a service, `sendBroadcast()` for BroadcastReceiver, etc.), whereas with AICC method we cannot statically directly know the type of those components (e.g., the signature of the `set()` method gives no information about the type of the target component, and it is the same for most of the AICC methods such as `sendTextMessage()`, `requestLocationUpdates()`, etc.).

Depending on the program's control flow during execution, the target component can change, hence its type too. Consequently, we have to take into account all possible types of different components. The main idea of our instrumentation approach is to add as many new standard ICC method calls as there are target components types and `Intent` objects for creating `PendingIntent` and `IntentSender` right after the program points of interest. The type is represented by a single character in the COAL specification for a given class. For example, the target type of a `PendingIntent` can take the following values: 1) "a" for an `Activity`, 2) "r" for a `BroadcastReceiver` and 3) "s" for a `Service`.

Step 3: After retrieving the possible target component types of the AICC methods,

RAICC has to recover the right `Intent` that has been used for creating the `PendingIntent` or the `IntentSender`, which will be the parameter of the generated standard ICC method(s). To tackle this issue, RAICC first recovers the `PendingIntent` or `IntentSender` reference used in the AICC method. Note that it can be used as a parameter in the AICC method (e.g., `sendTextMessage()`) or as the caller object (e.g., `send()`), we annotated each AICC method for having this information and, in the case it is a parameter, the index in the list of parameters. Afterwards, RAICC interprocedurally searches for the `Intent` used for creating the `PendingIntent`. In the case of `IntentSender`, RAICC interprocedurally searches for the `PendingIntent`, then recursively apply the previous process for retrieving the `Intent`. Of course, different `Intent` objects could be used in the code (not shown in Listing 10.4). Therefore for correctly propagating the "context information" among components for further analysis, they should all be taken into account, as RAICC does.
Step 4: At this point, for each point of interest (AICC method), RAICC leverages the list of potential target component types and the list of potential `Intents`. The source code modification of the app to explicitly set the ICC methods is straightforward. After each AICC method, RAICC generates as many invoke statements as there are combinations of potential target types and potential `Intents` recovered. The new generated invoke statements will depend on the type(s) inferred at *step 2*, i.e., `startActivity` for "a", `startService` for "s" and `sendBroadcast` for "r". `Intent` objects are used as parameters of the new method calls.

Note that some of the AICC methods, likewise `startActivityForResult()`, expect a result returned if the target component type is an `Activity`. We have carefully annotated the corresponding AICC methods, therefore RAICC generates the right method call in this case, i.e., `startActivityForResult()`.
Step 5: Finally, RAICC packages the newly generated application, and any existing tool dealing with standard ICC methods can be used to perform further static analysis.

Note that although instrumentation can lead to non-runnable apps, in this study, apps are not meant to be executed after being processed by RAICC. Indeed, RAICC acts as a preprocessor for other static analyses.

10.5 Evaluation

We address the following research questions:

RQ1: Do AICC methods deserve attention? In other words, are AICC methods often used in Android apps?

RQ2: Are AICC methods new in the Android Ecosystem?

RQ3: Can RAICC boost the precision of ICC-based data leak detectors on benchmark apps?

RQ4: Does RAICC reveal previously undetected ICC links in real-world apps? If so, are these newly detected ICC links security-sensitive?

RQ5: What are the runtime performance and the overhead introduced by RAICC?

10.5.1 Atypical ICC Methods Deserve Attention

In section 10.4.1, we described how we build a list of *atypical ICC methods*. We used this list to conduct empirical analyses assessing the use of AICC methods in the wild.

In the first study, we randomly selected 50 000 malicious apps and 50 000 benign apps from the Androzoo dataset [46]. For qualifying the maliciousness of the apps, we used

	Without libs			With libs		
Dataset	**# AICCM**	**# apps**	**ratio†**	**# AICCM**	**# apps**	**ratio†**
50k benign	124 226	24 884 (49.8%)	5/app	1 154 425	43 754 (87.5%)	26.4/app
50k malicious	402 468	34 710 (69.4%)	11.6/app	522 126	39 845 (79.7%)	13.1/app

†The ratio is computed by considering apps with at least one AICC method.

Table 10.1: Number of apps using at least one AICC method in different datasets (AICCM: AICC method).

the VirusTotal [55] score (number of antivirus products that flag an app as malicious) available in the metadata of the app in Androzoo. Every app of our malicious set has a VirusTotal score strictly greater than 20. Those from the benign have a score equal to 0.

Library code vs. developer code: It has been shown [56] than libraries present in Android apps can seriously impact empirical investigation performed on Android apps. Indeed, code related to libraries is often larger than the code written by the developers of the apps. For this reason, in this study, we perform two experiments: (1) we count the number of AICC methods present in each collected app by considering the entire code (i.e., including library code); (2) we count only the number of AICC methods present in the developer code. In practice, to exclude library code, we rely on Soot, which can discard third-party libraries from a given list (in our experiments, we use the list from [56]) and system classes with simple heuristics (e.g., discard if the signature starts with "androidx.*" or "org.w3c.dom.*", etc.)

Table 10.1 shows our findings. We can see that among the benign apps, considering only the developer code, 24 884 apps (∼50%) use at least one AICC method, and overall, 124 226 AICC methods are used. If we take into account the libraries, it is no less than 43 754 apps (87.5%) using in total 1 154 425 AICC methods. Clearly, in benign apps, the large majority of AICC methods are leveraged by libraries. In the malicious set, we face a different situation. The reported figures considering libraries or not, are much closer. Finally, if we compare both datasets, we note that overall, benign apps tend to use much more AICC methods than malicious apps, but when considering only the code written by the developers of the apps, the situation is reversed, i.e., developers use much more AICC methods in malicious apps than in benign apps.

Table 10.2 presents, for both datasets, the top 5 used AICC methods in developer code (excluding libraries). We notice 3 common AICC methods in this table (i.e., `set`, `setRepeating` and `setLatestEventInfo`). Regarding the malicious apps, we can see that the methods from the class `SmsManager` are present twice. It could be explained by the fact that malicious apps tend to activate components via SMS. We also note that method `setLatestEventInfo` is used an order of magnitude more than all other methods. This method is actually related to the notification mechanism of Android. We postulate that malicious apps tend to be much more aggressive in terms of notifications and advertisements, resulting in a high number of usages of this method.

Finally, Figure 10.3 presents the number of usages of each of the 111 AICC methods in the developer code in both benign and malicious datasets. For each dataset, the methods are ranked by their number of occurrences. For the sake of readability, we have truncated the first two bars of the malicious datasets. Indeed, as shown in Table 10.2, the number of occurrence of the top 3 methods are ∼238k, ∼53k and ∼39k respectively. Thanks to Figure 10.3, we note that: (1) only a fraction of the AICC methods is largely used by developers, (2) 21 methods are not even used at all, (3) malicious developers tend to use a less diverse set of AICC methods, but the AICC methods that are used are more frequently used.

Methods	Counts	%
Benigns (50 000)		
android.app.AlarmManager.set	27 214	21.9%
android.widget.RemoteViews.setOnClickPendingIntent	19 217	15.5%
android.app.Notification.setLatestEventInfo	18 024	14.5%
android.app.AlarmManager.setRepeating	9184	7.4%
android.app.Activity.startIntentSenderForResult	6876	5.5%
Malicious (50 000)		
android.app.Notification.setLatestEventInfo	238 462	59.2%
android.app.AlarmManager.set	53 533	13.3%
android.telephony.SmsManager.sendTextMessage	39 011	9.7%
android.app.AlarmManager.setRepeating	22 813	5.7%
android.telephony.SmsManager.sendDataMessage	13 075	3.2%

Table 10.2: Most used atypical ICC methods in benign/malicious Android apps, without considering libraries.

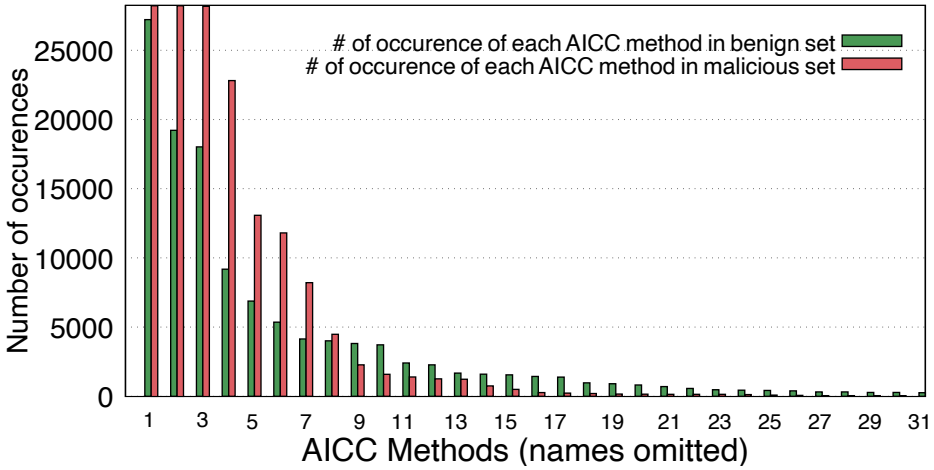


Figure 10.3: Occurence of AICC methods in benign and malicious apps (excluding libraries)

Dataset	Withtout libs # AICCM	Withtout libs # apps	Withtout libs ratio†	With libs # AICCM	With libs # apps	With libs ratio†
Benign Sets						
2016 (5000)	14 130	2620 (52.40%)	5.4/app	129 089	4584 (91.68%)	28.2/app
2017 (5000)	11 540	2486 (49.72%)	4.6/app	133 803	4601 (92.02%)	29.1/app
2018 (5000)	15 167	2487 (49.74%)	6.1/app	143 009	4708 (94.16%)	30.4/app
2019 (5000)	15 923	2629 (52.58%)	6.0/app	144 467	4528 (90.56%)	31.9/app
2020 (5000)	15 300	2403 (48.06%)	6.4/app	106 019	3488 (69.76%)	30.4/app
Malicious Sets						
2016 (5000)	20 156	2371 (47.42%)	8.5/app	58 967	2997 (59.94%)	19.7/app
2017 (2825)	16 316	1222 (43.26%)	13.3/app	45 832	1583 (56.03%)	28.9/app
2018 (3067)	28 083	1676 (54.65%)	16.8/app	56 623	1823 (59.44%)	31.1/app
2019 (548)	1494	378 (68.98%)	3.9/app	7268	429 (78.28%)	16.9/app

†The ratio is computed by considering apps with at least one AICC method.

Table 10.3: Temporal evolution of the usage of AICC methods in benign and malicious apps.

RQ1 Answer: AICC methods are prevalent in Android apps and thus definitely deserve attention. They are used in both malicious and benign apps but significantly more by malicious developers. Only a fraction of the AICC methods is regularly used.

10.5.2 Atypical ICC Methods exist since the beginning

To the best of our knowledge, state-of-the-art approaches do not consider AICC methods. One of the reasons could be the fact that AICC methods have only been recently introduced in the Android framework. To validate this hypothesis, we further check the use of AICC methods over time. For this purpose, we considered 5 sets of 5000 benign apps from Androzoo (ordered by the creation date of the dex file), and 4 sets of malicious apps. Androzoo only contains a few malicious apps from 2019 and no malicious apps from 2020. Thus, the 2019 malicious set is reduced compared to the benign one, and there is no 2020 malicious set. The sets, their content, and the results of the analyses are provided in Table 10.3.

First, overall these results confirm the results of Table 10.1. For instance, in benign apps, AICC methods are mostly used in libraries. Malicious developers still use more AICC methods in their code, even if the difference between with or without libraries is less pronounced. Regarding temporal evolution, we note that in both datasets, the metrics are pretty stable, except maybe in 2019 -malicious set- which seems to be an outlier (weak ratio and high % of the number of apps). This could be explained by the low number of apps (548) collected for 2019.

To deepen our investigation of temporal evolution, we also study the "introduction time" of the 111 AICC methods. To that end, we count the number of AICC methods introduced at each Android API level. The results are presented in Figure 10.4. New AICC methods have been added at almost each API level (often between 1 and 5 per API level). We can see two peaks: one at API level 1 corresponding to the creation of the Android framework, and one at API level 28 corresponding to the introduction of AndroidX, a new set of Android libraries. It is noteworthy that only two AICC methods have been removed from the Android framework.

RQ2 Answer: AICC methods are not new in the Android framework. They indeed exist since the very beginning.

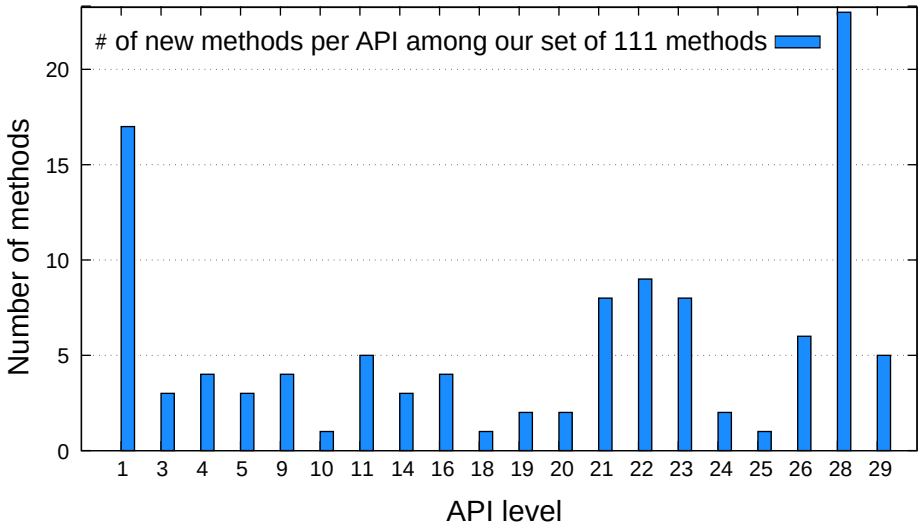


Figure 10.4: API levels in which AICC methods have been added.

10.5.3 Precision improvement after applying RAICC

RQ3 aims at investigating the efficiency of state-of-the-art ICC data leak detector IccTA and Amandroid after applying RAICC. To do so, we launched the tools before and after executing RAICC against 20 new apps that we plan to integrate into DroidBench [5], an open test suite containing more than 200 hand-crafted Android apps for evaluating the efficiency of taint analyzers. DroidBench is used as a ground truth by the research community in order to assess the efficiency of static and dynamic analyzers. It contains different types of leaks, e.g., intra-component, inter-component, inter-app, etc. However, among the ICC leaks, none of them uses AICC methods. Thus, our idea is to extend DroidBench with 20 additional test cases focusing on ICC leaks (concrete application of taint tracking) performed via AICC methods. Note that, to detect false positives, we included 4 apps without leaks among the 20 apps (i.e., only 16 apps contain a leak).

Benchmark construction To develop those 20 apps, we considered the most representative AICC methods for both malicious and benign apps identified in Section 10.5.1. More specifically, we considered the top 10 AICC methods (in terms of occurrences) in both datasets leading to 14 AICC methods (10+10-6 duplicates). We also randomly picked 4 additional AICC methods to reach the final number of 18 AICC methods (2 AICC methods have been used twice), which represent 93.5% and 91.1% of the AICC methods occurrences in our datasets of 50 000 benign apps and 50 000 malicious apps, respectively.

The implementation of most of our bench apps was straightforward and triggered the underlying inter-component communication. Excerpts of such bench apps are similar to the ones presented in Listing 10.3. However, some AICC methods have required more sophisticated code, e.g., those manipulating `Notification` objects, for instance the `addAction` AICC method. Another example of more complex bench app is related to the AICC method `setOnClickPendingIntent` of the `android.widget.RemoteViews` class. The `PendingIntent` set as a parameter of this method is triggered after the user clicks on a widget appearing on the device's home screen. The widget (declared in the AndroidManifest.xml file) has to be installed on the home screen before the user can click on it to trigger the target component.

Besides developing applications using AICC methods, we combine multiple aspects of how ICC can be performed. For example, in several apps, we considered the data flow within three different components or a data flow looping back into the first component to check the behavior or RAICC.

⊛ = true positive, ⋆ = false positive, ◯ = false negative, **C** = Components, UI = User Interaction

Test Case	**# C.**	**Leak**	**UI**	**IccTA**		**Amandroid**	
sendTextMessage1	2	•	○	◯	⊛	◯	⊛
setSendDataMessage	3	•	○	◯	⊛	◯	⊛
sendTextMessage2	2	○	○		⋆		⋆
addAction1	2	•	•	◯	⊛	◯	⊛
addAction2	2	○	•		⋆		⋆
requestNetwork	2	•	○	◯	⊛	◯	⊛
requestLocationUpdates	3	•	○	◯	⊛	◯	⊛
startIntentSenderForResult	2	•	○	◯	⊛	◯	◯
send	3	○	○				⋆
sendIntent	2	○	○				
setRepeating	2	•	○	◯	⊛	◯	⊛
setOnClickPendingIntent	3	•	•	◯	⊛	◯	⊛
setLatestEventInfo	2	•	•	◯	⊛	◯	⊛
setInexactRepeating	2	•	○	◯	⊛	◯	⊛
setExact	2	•	○	◯	⊛	◯	⊛
setExactAndAllowWhileIdle	2	•	○	◯	⊛	◯	⊛
setWindow	2	•	○	◯	⊛	◯	⊛
setDeleteIntent	2	•	•	◯	⊛	◯	⊛
setFullScreenIntent	2	•	•	◯	⊛	◯	⊛
setPendingIntentTemplate	3	•	•	◯	⊛	◯	⊛
Sum, Precision, Recall							
⊛, higher is better				0	16	0	15
⋆, lower is better				0	2	0	3
◯, lower is better				16	0	16	1
Precision $p = ⊛/(⊛+ ⋆)$				0%	88.90%	0%	83.33%
Recall $r = ⊛/(⊛+ ◯)$				0%	100%	0%	93.75%
F_1-score $= 2pr/(p+r)$				0	0.94	0	0.88

Table 10.4: Additional DroidBench apps and results of applying IccTA and Amandroid before and after RAICC.

Table 10.4 lists the 20 bench apps. We invite the interested reader to refer to the project repository[1], which contains the source code of each bench app.

Results Table 10.4 shows the results of our experiment. Since IccTA and Amandroid are not designed to detect ICC data leaks via AICC methods, it is not surprising to see that they perform very badly without applying RAICC (precision and recall of 0%). Indeed, IccTA and Amandroid are not able to construct the links between the components for the 16 apps containing a leak. However, the 4 apps which do not contain any leak do not raise any alarms as expected.

After instrumenting the apps with RAICC, the performance of IccTA and Amandroid is improved. They can reveal and construct previously hidden ICC enabling the detection of the leaks present in this benchmark.

Regarding IccTA, it is able to reveal all the leaks after applying RAICC. However, we can see 2 false positives. The first one, in app "sendTextMessage2", is due to IccTA which cannot correctly parse extra keys added into `Intent` objects (cf. `startActivity7` of DroidBench). The second one is due to RAICC, which cannot, for the moment, differentiate atypical inter-component communication made asynchronously. What we mean is that in "addAction2", the notification is never shown to the user, hence the component targeted by the `PendingIntent` will not be executed through the notification. Therefore, the leak cannot happen during execution. Even if declaring a notification and not showing it to the user is not likely to happen in practice, it is a good example to show that modeling an app behavior is not trivial and demands more effort for certain methods.

[1] `https://github.com/JordanSamhi/RAICC`

Component types	IC3 Counts	IC3 %	RAICC Counts	RAICC %	Increase %
Benign set (5000)					
Activity	17 095	84.2%	+2463	45.5%	+14.4%
BroadcastReceiver	1221	6.0%	+1907	35.3%	+156.2%
Service	1984	9.8%	+1038	19.2%	+52.3%
Total	20 300	100%	+5408	100%	+26.6%
Malicious set (5000)					
Activity	13 489	83.1%	+7340	73.4%	+54.4%
BroadcastReceiver	747	4.6%	+1468	14.7%	+196.5%
Service	1986	12.3%	+1193	11.9%	+60.1%
Total	16 222	100%	+10 001	100%	+61.6%

Table 10.5: Number of ICC links resolved by IC3 and number of additional ICC links discovered by RAICC.

We can notice that IccTA behaves correctly with apps "send" and "sendIntent" by not raising an alarm.

Amandroid performance is also boosted. Indeed, it can reveal almost all the leaks (1 false negative). We can notice that the same false positives appears for IccTA and Amandroid for apps "sendTextMessage2" and "addAction2". Amandroid reveals an additional false positive for app "send".

As a result, the precision of IccTA combined with RAICC reaches 88.90% (16 true positives and 2 false positives) and its recall 100% (16 true positives and 0 false negatives). As for Amandroid, combined with RAICC its precision reaches 83.33% (15 true positives and 3 false positives) and its recall is 93.75% (15 true positives and 1 false negative). IccTA F_1-score reaches 0.94 and Amandroid 0.88.

RQ3 Answer: RAICC boosts both the precision and the recall of state-of-the-art data leak detectors.

10.5.4 Experimental results on real-world apps

In this section, we first investigate to what extent RAICC discovers previously undetected ICC links in real-world apps. Then, we perform two checks on these newly detected ICC links: (1) we check if they are used to transfer data across components or even to perform some privacy leaks; (2) we check if they lead to ICC vulnerabilities.

Revealing new ICC links

In this section, we study the capacity of RAICC in revealing new ICC links in real-world apps. To that end, we extract, from Androzoo, two datasets of 5000 randomly selected apps containing respectively only benign and malicious apps. Then for each app, we count the number of ICC links discovered without RAICC (by relying on the results yielded by IC3), as well as the number of additional ICC links discovered by RAICC. Note that we only consider the developer code in this study (i.e., we exclude the libraries). Table 10.5 presents our results.

Among 5000 benign apps, 5408 new ICC links were revealed by RAICC, corresponding to an increase of more than 25% in comparison with IC3. The most used target component type is `Activity` with 45% of the new links. However, while for IC3 the large majority of ICC links are related to `Activity`, the distribution among the 3 types of component

is more balanced with RAICC. With regard to malicious apps, while the number of ICC links revealed by IC3 is relatively close to the number of ICC links revealed in the benign dataset (20 300 vs. 16 222), the number of ICC links revealed by RAICC is much higher, almost twice as much as benign apps (5408 vs. 10 001). Overall, RAICC increases the number of ICC links by 61% in malicious apps.

All three types of components are impacted by RAICC. However, the increase in the number of ICC links is impressive for `BroadcastReceiver`: 156% for benign apps and almost 200% for malicious apps. This suggests that developers tend to use AICC methods more than traditional ICC methods to "broadcast" an event. Through manual inspection, we indeed notice that, for instance, an AICC method attached to an "alarm" is often used to trigger a `BroadcastReceiver`.

Finally, note that we also randomly picked 40 benign and malicious apps to manually verify if RAICC had correctly instrumented the real-world apps. The standard ICC methods are correctly added right after the AICC methods, allowing other tools to model ICC correctly.

Atypical ICC methods are largely used in real-world apps, although not to transfer or leak data

For this study, we only consider a set of 5000 malicious apps (the underlying intuition is that malicious apps tend to leak more data than benign apps). We first run RAICC on this dataset (to resolve the atypical ICC links), and then we leverage IccTA to perform the detection of ICC leaks (IccTA uses a set of well-defined sources (i.e., sensitive information) and sinks to perform the detection). Overall, IccTA was able to detect 6129 intra-component data leaks (i.e., leaks inside a single method) and 114 ICC data leaks. We manually inspect all 114 ICC data leaks to check if the data is transferred via AICC methods or standard ICC methods such as `startActivity()`. We did not find a single case where sensitive information was leaked via AICC methods.

We manually analyzed 60 apps to verify how AICC methods were used. In the majority of cases, the target component is used like a callback method, i.e., this mechanism is used to "activate" a given component. Actually, when data is put inside the `Intent` used for constructing the `PendingIntent` or the `IntentSender`, it is generally non-sensitive data (most of the time simple constants). Let us consider a concrete example, for instance, the "M1 Trafik" app from the Google PlayStore. In method `setAlarm` of class `com.m1_trafik.AlarmManagerBroadcastReceiver`, an `Intent` is created with an extra value representing the `Boolean` false value. Information attached to this intent also informs us that the target component is the current class itself (i.e., the class `AlarmManagerBroadcastReceiver`). A `PendingIntent` is then retrieved from this `Intent` using method `getBroadcast()`. Afterwards, the AICC method `setRepeating()` of class `AlarmManager` is leveraged for setting an alarm. When this alarm goes off, the method `onReceive` of the target component (in our case, the same class) is executed. When analyzing this method we can see no use of the extra value put in the `Intent`. When applying RAICC, we can see the new method call `sendBroadcast()` right after the call to `setRepeating()`. Although it helps IccTA construct the link between the components, the data transferred is not sensitive. In this example, we see that AICC methods are mostly used to leverage the powerful "token" mechanism explained in Section 10.3, i.e., the target component will be launched even if the application is closed.

RAICC & EPICC: revealing new ICC vulnerabilities

EPICC [138] is a state-of-the-art ICC links resolver able to detect ICC vulnerabilities. Such vulnerabilities are defined by Chin et al. in [134]. Examples include (1) when

	1000 benign apps	1000 malicious apps
Before RAICC	4796	9544
After RAICC	5032	9868
Improvement	+236 (+4.9%)	+324 (+3.4%)

Table 10.6: Number of ICC vulnerabilities found by EPICC before and after applying RAICC

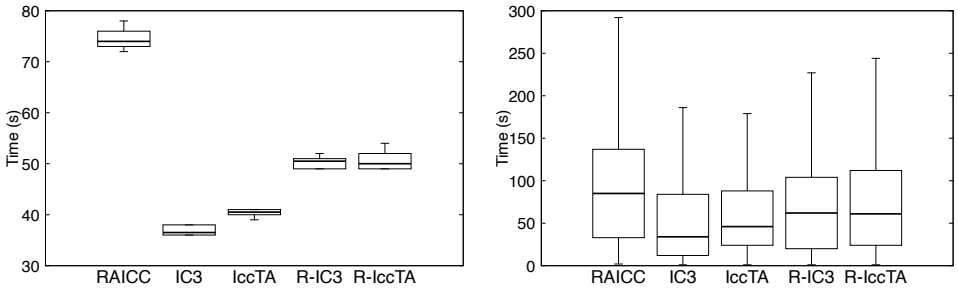


Figure 10.5: Runtime performance of RAICC, IC3, and IccTA with (R- means with RAICC) and without AICCM preprocessing. (left: on Droidbench, right: in the wild).

an app sends an Intent that may be intercepted by a malicious component or (2) when legitimate app components –e.g., a component sending SMS messages– are activated via malicious *Intent*. In this section, we aim to show that RAICC boosts EPICC by enabling the detection of previously unnoticed ICC vulnerabilities. To this end, we considered a dataset of 1000 randomly selected benign apps and a dataset of 1000 randomly selected malicious apps. We ran EPICC on those two datasets before and after applying RAICC. Results are available in Table 10.6.

Besides the significant difference between benign and malicious apps, we can see that after applying RAICC, i.e., modeling previously unrevealed ICC links, EPICC is able to detect more ICC vulnerabilities, with an increase of 4.9% for benign apps and 3.4% for malicious apps. This experiment shows that RAICC boosts state-of-the-art tool EPICC by modeling new ICC links and revealing new ICC vulnerabilities.

RQ4 Answer: RAICC significantly increases the number of resolved ICC links in real-world apps compared to the state-of-the-art approach. While AICC methods do not seem to leak sensitive information, they are used to activate components (and thus potentially trigger malicious payloads). RAICC boosts EPICC by allowing to reveal new ICC vulnerabilities.

10.5.5 Runtime performance of RAICC

In this section, we evaluate the runtime performance of RAICC. We also evaluate the overhead introduced by our tool by considering a typical usage of RAICC, for instance, when RAICC is used to boost the results of IccTA. Since IccTA leverages itself IC3, we investigate the runtime performance of IC3 and IccTA before and after applying RAICC on the 10 benchmark apps used in Section 10.5.3.

The results are presented in Figure 10.5. First, we can see that the RAICC execution time does not exceed 80 seconds. Since RAICC allows IC3 and IccTA to resolve additional ICC links, we expect that the analysis time of both tools will increase. We note that the two box plots on the right are higher, confirming the overhead caused by RAICC. On average, the overheads for IC3 and IccTA are 13.3 seconds and 10 seconds respectively (36.74% and 24.74% overhead respectively).

To confirm the results obtained on the benchmark apps, we performed the same study but on a set of 1000 real-world apps. The results are reported in Figure 10.5. Overall, we can see that both figures' performances (in time) are pretty similar. However, slight differences can be noticed. First, the runtime values are more scattered in Figure 10.5 than with the benchmark apps. This could be explained by the fact that real-world apps are more diverse. Second, the average performances of the three tools are closer.

Regarding the overhead introduced by RAICC, we again notice that this overhead exists. This is expected since the constant propagation of IC3 has to process more values/methods. Likewise, IccTA has to build more links and consider more paths for the taint analysis. On this dataset, on average, the overheads for IC3 and IccTA are 21.8 seconds and 5.8 seconds respectively (+43.8% and +6.5% overhead respectively).

RQ5 Answer: The runtime performance of RAICC is higher than IC3 and IccTA, but still in the same order of magnitude. On average, RAICC requires less than 2 minutes to analyze and instrument a real-world application.

10.6 Limitations

The core component of our approach lies in the list of AICC methods that we compiled during our research. Even though we followed a systematic approach for retrieving a maximum of AICC methods, we might have missed some of them in the Android Framework. There are potentially other ways to perform such ICC. Nevertheless, our study is reproducible and provides insight for future research in this direction.

By leveraging IC3 to infer the values of ICC objects, RAICC inherits the limitations of IC3. Moreover, like most of the static analysis approaches, RAICC is subject to false positives. Currently, RAICC does not handle native calls, reflective calls or dynamic class loading, though some state-of-the-art approach could be integrated [148, 8]. Besides, although inter-app communication (IAC) is performed using the same mechanisms as ICC [149], we did not investigate in this direction.

Furthermore, obfuscation is a confounding factor impacting studies based on APKs [150, 151]. Therefore, RAICC's effectiveness is impacted by obfuscated code, especially if AICC method calls are disguised (e.g., using reflection).

10.7 Related work

To the best of our knowledge, we have presented the first approach taking into account AICC methods for connecting Android components. However, as explained in a systematic literature review [104], the research literature has proposed a large body of works focusing on statically analyzing Android apps. One of the most popular topics is the use of static analysis for checking security properties, and in particular for checking data leaks. The pioneer tools such as FlowDroid, Scandal, and others [5, 96, 152, 153, 154] have started to focus on the detection of intra-component data leaks. They all face the limitations of not being able to detect ICC leaks.

Several approaches have been developed to perform data leak detection between components. We will present these approaches in the following. **IccTA** [6] leverages IC3 [139] to identify ICC methods and theirs parameters, and then instruments the app by matching and connecting ICC methods with their target components. The identification of ICC methods and the instrumentation part rely on a list of ICC methods that only contain *well-documented ICC methods*. By considering additional ICC methods (i.e., AICC methods), our tool complements a tool such as IccTA. In the same way, **DroidSafe** [135] transforms ICC calls into appropriate method calls to the destination component. Likewise, IccTA,

the ICC methods considered by DroidSafe are only the well-documented ICC methods. As a result, both DroidSafe and IccTA share the same limitation, i.e., they miss the AICC methods. Unlike the previously described tools, **Amandroid** [11] constructs an inter-component data flow graph (IDFG) and a data dependence graph (DDG) in which it can run its analysis. Again, it only considers *documented ICC methods* manipulating `Intents`.

Other tools leverage ICC links to detect malicious apps. ICCDetector [137], for instance, uses Machine Learning (ML) to detect Android malware The ML model is built by using ICC-related features extracted with EPICC [138]. As it relies on EPICC to extract ICC features, it is dependent on EPICC for the considered ICC methods. Yet, EPICC, just as IC3 only considers *documented ICC methods* for inter-component communication. In the same way, Li & al. [155] set up an ML approach for detecting malicious applications. The feature set used is based on Potential Component Leaks (PCL) in Android apps. PCLs are defined using components as entry and/or exit points. Again, they consider traditional ICC methods as exit points for transferring data through components.

ICCMATT [156] aims at conceptually modeling ICC in Android apps to generate test cases. The purpose is to identify components vulnerable to malicious data injection and privacy leaks. The approach of the researchers takes into account `PendingIntent` objects, but only at the conceptual level. They describe them as `Intent` wrappers able to be shared between components, mainly used in notifications and/or alarm services, as we have seen throughout this work. They do not directly refer to methods for performing inter-component communication atypically with `PendingIntent` objects.

In the same way, Enck et al. [157] describe the overall functioning of `PendingIntents` for integration with third-party applications. Nevertheless, they do not explain, as in [145], the security threats it poses and the difficulty it induces for ICC modeling in static analyzers. PiAnalyzer [145] models specific vulnerabilities where other apps can intercept broadcasted PendingIntents. In contrast, RAICC generically models ICC links where PendingIntent (as well as IntentSender) are involved. The goals of PiAnalyzer and RAICC are thus different. Hence, RAICC was not compared to PiAnalyzer in this study.

Besides static analysis approaches, dynamic analysis solutions have also been studied for the detection of ICC data leaks. For example, **CopperDroid** [119] is able to reconstruct the app behavior by observing interactions between the app and the underlying Linux system. **TaintDroid** [16] dynamically tracks sensitive information with a modified Dalvik virtual machine. Monitoring the behavior of an Android app is also popular in dynamic data leak detection [158, 159, 160, 161]. Depending on the taint policy for propagating tainted data, a dynamic analysis could be considered and therefore detect atypical ICC data leaks. Nonetheless, precise methods exist [162] for bypassing taint-tracking, leading to false negatives as well as more general approaches for tackling ICC-related security issues [163, 164].

10.8 Summary

We addressed the challenge of precisely modeling inter-component communication in Android apps. After empirically showing that Android apps can leverage atypical ways of performing ICC, we discuss the implications for state-of-the-art ICC modeling-based analysis. We contribute towards using methods not primarily made for this purpose. We have developed and open-sourced RAICC, which reveals AICC methods and further resolves them into standard ICC through instrumentation. We demonstrate that RAICC can boost existing analyzers such as Amandroid and IccTA, enabling them to substantially increase their data leak detection rates.

Chapter 11

A Step Towards Android Code Unification for Enhanced Static Analysis

In this chapter, we propose a new approach to account for native code in Android apps which is currently overlooked in the literature. This limitation of the state of the art is a severe threat to validity in a large range of static analyses that do not have a complete view of the executable code. To address this issue, we propose a new advance in the ambitious research direction of building a unified model of all code in Android apps. Our approach, JuCify*, is a significant step towards such a model, where we merge both representations at the call graph and instruction levels. We performed empirical investigations to demonstrate how* JuCify *improves apps' call graph and static analyzers' results.*

This chapter is based on our work published in the following research paper:

- **Jordan Samhi**, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis. *In Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (**ICSE**)*. IEEE, 2022, 10.1145/3510003.3512766 [27].

Contents

11.1 Overview

Android app analysis has been one of the most active themes of software engineering research in the last decade. Static analysis research, in particular, has produced a variety of approaches and tools that are leveraged in a variety of tasks, including bug detection, security property checking, malware detection, and empirical studies. The widely-used state-of-the-art approaches, such as FlowDroid [5], develop analyses that focus on the Dex bytecode in apps. Unfortunately, recent studies [165, 166, 118, 167, 119] have shown that malware authors often build on native code to hide their malicious operations (e.g., private data leak) or to implement sandbox evasion [84].

The need to account for native code within Android apps is becoming urgent as the usage of native code is growing within both benign and malicious apps. Our empirical investigation on apps from the AndroZoo [46] repository reveals that, in 2019, up to 62.9% of collected apps included native code within their packages. Yet, native code is scarcely considered in app security vetting [166, 102]. In the majority of static [2, 17, 4, 44, 70, 5, 6], dynamic [12, 168, 14] and machine learning based techniques [73, 72], native code is overlooked since it presents several challenges.

When researchers propose techniques to address native code such as with *JN-SAF* [166], *DroidNative* [102], *NativeGuard* [169], *TaintArt* [170] and others [171, 167, 165, 172], the integrated analyses (e.g., for taint tracking, native entry point detection and machine learning feature extraction) are generally ad-hoc. Indeed, these works develop custom techniques to bridge native code and bytecode, typically by combining the results of separate analyses of bytecode and native code. Therefore, they do not yield an explicit unified model of the app to which generic analyses can be applied to explore bytecode and native code altogether.

Our work aims to fill the gap in whole-app analysis by researching means to build a unified model of Android code. We propose JuCify, a step toward building a framework that breaks bytecode-native boundaries for Android apps and therefore copes with a common limitation of static approaches in the literature. To the best of our knowledge, JuCify is the first approach that targets the unification of Android bytecode and native code into a unified model and is instantiated in a standard representation [104]. We target the Jimple [45] Intermediate Representation as support for JuCify unified model. Jimple is the internal representation in the widely-used Soot framework and is indeed the representation that is considered in a large body of static analysis works [104]. By supporting Jimple, JuCify provides the opportunity for several analyses in the literature to readily account for native code.

JuCify is a multi-step static analysis approach that we implement as a framework for generating a unified model of apps considering native code. It ① relies on symbolic execution to retrieve invocations between both the Dex bytecode and the native worlds, ② pre-computes native call graph, ③ merges Dex bytecode and native call graphs, and ④ populates newly generated functions with heuristic-based defined Jimple statements using code instrumentation.

The main contributions of our work are as follows:

- We propose JuCify, an approach to build a unified model of Android app code for enabling enhanced static analyses. We have implemented JuCify to produce the Jimple code that unifies bytecode and native code within an app package;
- We conduct an assessment of the JuCify yielded model. We show that JuCify can significantly enhance Android apps' call graphs. JuCify connects previously unreachable methods in Android apps' call graphs;
- We evaluate the unified model of app code in the task of data flow tracking. We show that JuCify can significantly boost the precision of the state-of-the-art Flowdroid, from 0% to 82% and its recall from 0% to 100% on a new benchmark targeting

bytecode-native data flow tracking;
- We evaluate JuCify on a set of real-world Android apps and show that it can augment existing analyzers, enabling them to reveal sensitive data leaks that pass through the native code which were previously undetectable.
- We release our open-source prototype JuCify to the community as well as all the artifacts used in our study at:

`https://github.com/JordanSamhi/JuCify`

11.2 Background & Motivation

Java and Kotlin are the two mainstream programming languages that support the development of Android apps. Their codes are compiled into Dex bytecode and included within app packages (in the form of DEX files). Nevertheless, thanks to the *Java Native Interface* [173], native code functionalities are accessible in Android apps. They come in binary (e.g., `.so` shared library) files compiled from input programs written in C/C++ for instance.

11.2.1 Java Native Interface (JNI)

JNI is an implementation of the *Foreign Function Interface* (FFI) [174] mechanism that allows programs written in a given language to invoke subroutines written in another language. JNI allows both Java to native and native to Java invocations.

Java to native code

Listing 11.1 presents an example where JNI capabilities are used to call a native function (here written in C++) from Java. First, a relevant Java method is defined with the keyword `native` (line 4). We will refer to it as a *Java native method*. Then, its corresponding native function is registered to set up the mapping between them. Such registration can be:
Static - the native function definition follows a naming convention based on specific JNI macros. For example, the Java native method `nativeGetImei` (line 4) corresponds to a native function named `Java_com_example_nativeGetImei` in C++ (line 16).
Dynamic - developers can arbitrarily name their native functions (in C++) as shown in Listing 11.2 (lines 10–13) but must inform JNI about how to map them with Java native methods. Thus, developers ① first map Java native methods to their counterpart native functions by using specific `JNINativeMethod` structures (lines 14-16 in Listing 11.2); ② overload a specific JNI Interface function [175], `JNI_OnLoad`, to register the mapping (lines 17–24 in Listing 11.2); and ③ invoke `RegisterNatives` in `JNI_OnLoad` which will be called by the Android virtual machine (line 22 in Listing 11.2).

Native to Java

With JNI, developers can create and manipulate Java objects within the native code (e.g., written in C++). The fields and methods of Java objects are also accessible from the native code and can be invoked using specific JNI *Interface* functions. Eventually, likewise Java reflection [176], i.e., using strings to get methods and classes, the developer can invoke the Java methods (e.g., lines 17–19 in Listing 11.1).

Note that Listings 11.1 and 11.2 illustrate the interaction between Java and C++. However, JuCify, the approach proposed in this study, works at the apk level. Therefore, the invocations are between bytecode and compiled native code.

```
/*** JAVA WORLD ***/
public class MainActivity extends Activity {
  static {System.loadLibrary("native-lib");}
  public native String nativeGetImei(TelephonyManager tm);
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    TelephonyManager tm =  (TelephonyManager) getSystemService("phone");
    String imei = nativeGetImei(tm);
    Log.d("IMEI",  "" + imei);
  }
  public void malicious() {/* malicious code */}
}
/*** C++ WORLD ***/
JNIEXPORT jstring JNICALL
Java_MainActivity_nativeGetImei(JNIEnv *env, jobject thiz, jobject tm) {
  jclass c = (*env).GetObjectClass(tm);
  jmethodID m = (*env).GetMethodID(c, "getImei", "()Ljava/lang/String;");
  jstring i = (jstring)(*env).CallObjectMethod(tm, m);
  c = (*env).GetObjectClass(thiz);
  m = (*env).GetMethodID(c, "malicious", "()V");
  (*env).CallObjectMethod(thiz, m);
  return i;
}
```

Listing 11.1: Code illustrating how an app can trigger native code. (Methods and code are simplified for convenience)

11.3 Approach

For a given Android app, JuCify aims to unify its Dex bytecode and native code into a unified model and instantiate this model in the Jimple representation (i.e., the intermediate representation of the popular Soot framework). In this section, we will first detail the overall JuCify conceptual approach, and then we will briefly present how we instrument the app to approximate the native behavior. We invite the interested reader to consider all our publicly-shared artifacts on the project GitHub's repository[1]. JuCify implementation is fully open-sourced.

11.3.1 Call Graph as Unified Preliminary Model

To explain the overall functioning of JuCify, we will restrict our explanations to the notion of Call Graph (CG). A CG can be defined as $CG = (V, E)$, where V is a set of vertices representing functions, and $E \subseteq \{(u, v) \mid u, v \in V\}$ is a set of edges such as $\forall (u, v) \in E$, there is a call from u to v in the program.

JuCify is a multi-step static analysis framework whose overall architecture is depicted in Figure 11.1. First, a submodule called NativeDiscloser constructs the native call graph and extracts the mutual invocations between bytecode and native code. Then, the native call graph is pruned and prepared to be Soot-compliant before being merged with the bytecode call graph. Eventually, both call graphs are unified thanks to information related to the bytecode-native method invocations. The following gives more details about the different steps of our approach.

Step 0: Native Call Graph Construction

Native program call graph construction is not trivial [177]. In fact, a large body of work tackled this problem and proposed several solutions to find function boundaries [177,

[1] https://github.com/JordanSamhi/JuCify

```
/*** JAVA WORLD ***/
public class MainActivity extends Activity {
  static {System.loadLibrary("native-lib");}
  public native String nativeMethod();
  @Override
  protected void onCreate(Bundle b) {nativeMethod();}
}
/*** C++ WORLD ***/
JNIEXPORT jstring JNICALL
jstring arbitrary_name(JNIEnv *e, jobject thiz) {
  std::string str = "str";
  return e->NewStringUTF(str.c_str());
}
static const JNINativeMethod m[] = {
  {"nativeMethod", "()Ljava/lang/String;", (jstring*)arbitrary_name}
};
JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved){
  JNIEnv* e = NULL;
  if(vm->GetEnv((void**)&e, JNI_VERSION_1_4) != JNI_OK){return -1;}
  jclass c = e->FindClass("com/example/MainActivity");
  if (!c){return -1;}
  if(e->RegisterNatives(c, m, sizeof(m)/sizeof(m[0]))){return -1;}
  return 1;
}
```

Listing 11.2: Dynamic native function registration example. (Methods and code are simplified for convenience)

178, 179]. In this work, the native libraries' call graphs in Android apps are generated by Angr [180], a well-known binary analysis framework wrapped into our submodule NativeDiscloser.

Step 1: Bytecode-Native Code Invocations Extraction

This step is performed over 4 sub-steps: ① retrieve bytecode methods information; ② extract entry method invocations (i.e., bytecode to native); ③ track native function calls and extract exit method invocations.

Step 1.1: Methods info extraction is a straightforward task that extracts information about bytecode methods, such as the class of a method and the method signature. This step aims to complete the signature information required to perform the method invocations extraction task for statically registered functions. We perform this task by relying on AndroGuard [181].

Step 1.2: Entry method invocations extraction: An entry method invocation is a native method invocation from the bytecode (i.e., a bytecode-to-native "link"). As described in Section 11.2.1, for such an invocation, we need to match a "Java native method" (i.e., a method declared in Java with the `native` keyword, also called *entry method*) and an *entry function* (i.e., the counterpart native function). To perform this task, we must take care of static and dynamic registrations. The statically registered functions can be easily spotted via their naming conventions. However, more sophisticated techniques are required as dynamic registration relies on JNI *interface* function calls. In our case, we rely on symbolic execution.

From a more technical point of view, NativeDiscloser takes as input the library (i.e., `.so`) files of an apk and the method information from the previous step. It first scans the symbol table of each binary to search for (1) statically registered native functions and (2) the `JNI_OnLoad` function for the case of dynamically registered functions. Then, if `JNI_OnLoad` exists, this function is symbolically executed to further detect dynamically registered native functions.

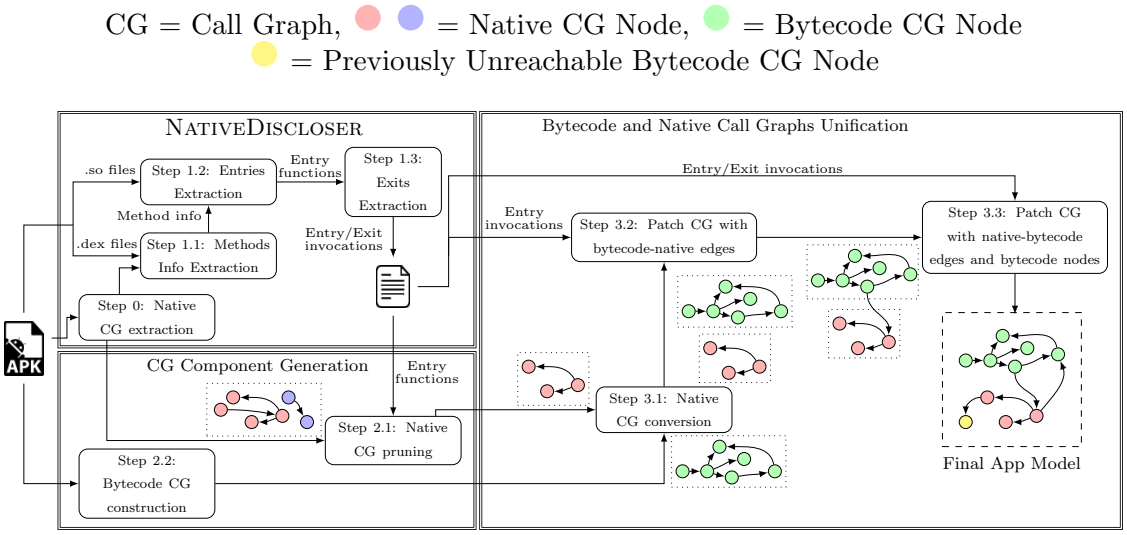


Figure 11.1: Overview of the JuCify Approach from the Angle of Call Graph Construction

For symbolic execution, NativeDiscloser relies on Angr [180].

Step 1.3: Exit method invocations extraction: We are looking for the invocations of a bytecode method from the native code. We call *exit method* this bytecode method. In Section 11.2.1, we explained that this exit method is called by invoking certain JNI *Interface* functions in a chained manner. Collecting information related to this chain of JNI function invocations is challenging.

In practice, to overcome this challenge, NativeDiscloser executes all the entry functions acquired from step 1.2 symbolically to search for the exit method invocations and set up the relation mapping between entry and exit method invocations.

Furthermore, exit methods could be invoked deep down in a native function chain. However, the symbolic execution is unaware of the boundaries between native functions. Hence, we implemented a tracking mechanism during the search for exit methods. We rely on the starting address of each native function obtained from the native call graph to maintain a stack of native functions and push a new function into the stack when its starting address is reached. Popping a function from the stack is triggered by the arrival of the return addresses of native functions, which can be obtained from a certain register or memory location based on architecture specifications (e.g., link register *LR* for *ARM*) during entering a native function. This allows us to know from which native function an exit method invocation occurs.

Step 2: CG Components Generation

Step 2.1: Native CG pruning. Since in .so libraries, not all the functions are necessarily called in an app, we rely on a strategy to only keep relevant call graph parts. To do so, we prune the obtained native call graphs constructed in Step 0 with the help of the entry functions passed in from Step 1. We only keep the sub-graphs starting from the entry functions (with all successor nodes) since the remaining parts will not be reachable from the bytecode.

Step 2.2: Bytecode CG construction. Our approach also requires the bytecode call graph. For this purpose, we use Flowdroid [5] (itself based on Soot [24]) which leverages an advanced modeling of app components' life-cycle.

Step 3: Bytecode and native call graphs unification

Step 3.1: Native CG conversion. In practice, the target is to load both native and bytecode call graphs in Soot. Although this is straightforward for the bytecode call graph, the native call graph requires a conversion step to fit with Soot technical constraints. Once loaded, the sets of nodes and edges of both call graphs are merged, but the call

graphs are not yet connected together.

Step 3.2: Patch CG with bytecode-to-native edges. Then, according to the entry invocations obtained from Step 1.2, edges between entry methods (in bytecode) and their counterpart entry functions (in native code) are added.

Step 3.3: Patch CG with native-to-bytecode edges and bytecode nodes. Finally, with the information on exit invocations and the relations with entry invocations from Step 1.3, edges between native functions to exit methods are added. This step allows to uncover previously unreachable bytecode call graph nodes.

11.3.2 From CG to Jimple for a Unified Model

A call graph is a useful model, but it is still limited because it does not contain enough information to perform static analysis (e.g., data flow analysis). Indeed, important information, such as the statements present in each method, is missing (i.e., the control flow graph (CFG)). A tool such as FLOWDROID provides the CFG for each bytecode method, representing the method behavior with Jimple statements. We will now explain how JUCIFY adds Jimple statements in specific native functions in a best-effort mode. After this step, for a given APK, we obtain the Jimple representation of the apk with both bytecode and native code unified.

Native functions generation: JUCIFY relies on a *DummyBinaryClass* whose purpose is to incorporate any newly imported native function in the SOOT representation. For each native function in the native call graph, JUCIFY generates a new method in the *DummyBinaryClass* with appropriate signatures.

Bytecode method statements instrumentation: JUCIFY generates bytecode-to-native call graph edges. It also has to replace the initial call to the native method at the statement level with a call to the newly generated native function. JUCIFY takes care of the returned value and the parameters to not fool any analysis based on the newly built model.

Native function statements generation: There is no bijection between native code and JIMPLE code [45]. Moreover, bytecode and native code manipulate different notions (e.g., pointers) that cannot be translated directly. Therefore, we have to use heuristics based on the information at our disposal to put a first step toward reconstructing native function behavior.

Let us consider a native function named `foo()` containing at least one invocation to a bytecode method m. As explained in Section 11.3.1, the first step of JUCIFY aims to collect information about bytecode methods (full signature). Thanks to this, we can approximate the parameters used by m as well as its return values.

More specifically, in Listing 11.3, we detail the steps JUCIFY implements to populate the native function `foo()` that calls a bytecode method m. Let consider m is defined in a Java class named `MyClass`. In line 1, JUCIFY starts with the empty method `foo()`. Then:

Step 1 in Listing 11.3: If the bytecode method m should return a value, JUCIFY generates a new local variable with the same type as the method's return type (line 4).

Step 2 in Listing 11.3: JUCIFY generates the declaration of a variable of type `MyClass`, the class in which m is defined (line 8). In line 9, JUCIFY creates a new `MyClass` instance (if there is not one usable as a base for the bytecode call).

Step 3 in Listing 11.3: Regarding the parameters that should be used for the invocation of m, JUCIFY scans `foo()` for local variables and parameters whose types match the types of the parameters of m. If, for a given type, no local variable, nor parameter of `foo()` is found, JUCIFY generates one (e.g., line 15). Then, it generates all the permutations of these variables with a given length (i.e., the number of parameters of m) and retains only those matching the types' order of the parameters of m ($(i1, s)$, and $(i2, s)$ in Listing 11.3). Each retained permutation corresponds to a possible call to the bytecode method in the

```
1  public boolean foo(int i1, int i2, boolean b1){}
2  // STEP 1
3  public boolean foo(int i1, int i2, boolean b1){
4    boolean b2;
5  }
6  // STEP 2
7  public boolean foo(int i1, int i2, boolean b1){
8    boolean b2; MyClass jc;
9    jc = new MyClass();
10 }
11 // STEP 3
12 public boolean foo(int i1, int i2, boolean b1){
13   boolean b2; MyClass jc; String s;
14   jc = new MyClass();
15   s = new String();
16   if(opaque_predicate) {
17     b2 = jc.m(i1, s);
18   }
19   else if(opaque_predicate) {
20     b2 = jc.m(i2, s);
21   }
22 }
23 // STEP 4
24 public boolean foo(int i1, int i2, boolean b1){
25   boolean b2; MyClass jc; String s;
26   jc = new MyClass();
27   s = new String();
28   if(opaque_predicate) {
29     b2 = jc.m(i1, s);
30   }
31   else if(opaque_predicate) {
32     b2 = jc.m(i2, s);
33   }
34   if(opaque_predicate) {
35     return b1;
36   }
37   else if(opaque_predicate) {
38     return b2;
39   }
40 }
```

Listing 11.3: JuCify's process to populate native functions

native function as an over-approximation. Nevertheless, these calls cannot be generated sequentially since they correspond to different realities. Hence, we rely on opaque predicates (*if* statements whose predicate cannot be evaluated statically) so that each control flow path is considered identically (lines 16–21).

Step 4 in Listing 11.3: If the native function returns a value (from the signature of `foo()`), JuCify should generate return statements. To do so, it operates as for m. It relies on opaque predicates. Indeed, first, JuCify scans the body of the current native function to find any local variable corresponding to the return value type (even those newly generated local variables that could be returned). If no variable is found, JuCify generates such a variable. Else, for each of found local variables, JuCify generates return statements with opaque predicates so that each path can be equally considered (lines 34–39 in Listing 11.3).

Finally, JuCify yields a unified model of Android apps on which analysts can perform any static analysis.

	Goodware			Malware		
	# Apps	w/ .so files	w/ native methods	# Apps	w/ .so files	w/ native methods
2015	632 279	220 934 (34.9%)	216 329 (34.2%)	89 542	65 221 (72.8%)	63 275 (70.7%)
2016	1 103 899	405 209 (36.7%)	404 357 (36.6%)	48 358	35 601 (73.6%)	34 240 (70.8%)
2017	277 690	143 463 (51.7%)	143 183 (51.6%)	15 141	8742 (57.7%)	8539 (56.4%)
2018	304 746	191 491 (62.8%)	184 447 (60.5%)	10 890	8415 (77.3%)	8018 (73.6%)
2019	179 309	113 433 (63.3%)	112 873 (62.9%)	9773	8993 (92.0%)	8311 (85.0%)
2020	143 271	81 755 (57.1%)	81 111 (56.6%)	638	446 (69.9%)	274 (42.9%)
Total	2 641 194	1 156 285 (44%)	1 142 300 (43%)	174 342	127 418 (73%)	122 657 (70%)

Table 11.1: Number and proportion of Android apps that contain at least one "`.so` file" / "Java native method" (w/ = with).

11.4 Evaluation

We investigate the following research questions to assess the importance of our contributions:

RQ1: What is the proportion and evolution of native code usage in both real-world benign and malicious apps?

RQ2: To what extent our bytecode-native invocation extraction step (named NativeDisCloser) yields better results than the state of the art?

RQ3: Can JuCify boost existing static data flow analyzers?

RQ4: How does JuCify behave in the wild? We address this question both at the quantitative and qualitative levels:

- **RQ4.a:** To what extent can JuCify augment apps' call graphs and reveal previously unreachable Java methods?
- **RQ4.b:** Can JuCify reveal previously unreachable data leaks that pass through native code in real-world apps?

11.4.1 RQ1: Native code usage in the wild

This section presents general statistics about the usage of native code in both benign and malicious Android apps. We also perform an evolutionary study of this usage.

Dataset: We rely on the AndroZoo repository [46] to build ① a dataset of 2 641 194 benign apps (where we consider an app as benign if no Antivirus in VirusTotal [55] has flagged it - score 0); and ② a dataset of 174 342 malicious apps (where we consider an app as malicious when at least 10 Antivirus engines in VirusTotal have flagged it). Both datasets contain all the apps from 2015 to 2020 that we were able to collect from AndroZoo with the mentioned VirusTotal constraints.

Empirical study: Android programming with the Native Development Kit (NDK) suggests developers to integrate native libraries (i.e., `.so` files) whose code can be invoked from the Java world. Therefore, to study the extent of native code usage in Android apps, as a preliminary study, for each app, we check if it contains at least one .so file in its APK file. However, since native libraries can be present in apps but never used, we also check for each app if Java native methods (cf. Section 11.2.1) are declared in the bytecode.

Results of our empirical study are presented in Table 11.1. They indicate that, overall, 1 156 285 benign apps (i.e., 44%) contain at least one .so file, and 1 142 300 (i.e., 43%) contain at least one Java native method declaration. This means that 98.8% of apps with native libraries contain Java native method declaration in their bytecode. Regarding malware, 127 418 (i.e., 73%) of apps contain native libraries and 122 657 (i.e., 70%) Java

TP = True Positive, FP = False Positive, FN = False Negative

Benchmark	Native-Scanner			NativeDiscloser		
	TP	FP	FN	TP	FP	FN
bm_{1-5}, bm_7, bm_{10-12}[†]	1	0	1	2	0	0
bm_6, bm_8	1	0	2	3	0	0
bm_9	0	0	2	0	0	2
bm_{13}	0	1	5	5	0	0
bm_{14}	1	4	1	2	0	0
bm_{15}, bm_{16}	1	0	1	1	0	1
Precision			73.68%			100%
Recall			37.84%			89.19%

Table 11.2: Comparison of Tools

native method declarations. Hence, 96.3% of malware with native libraries contain Java native method declarations. Overall, these results show that native code is, in proportion, more used in malicious apps.

Regarding usage evolution in benign apps, the rate increases until 2018 to reach a plateau at around 60%. The trend regarding malware is much more erratic (with sharp decreases in 2017 and 2020). However, for each year, malicious apps use significantly more native code than benign apps.

RQ1 answer: Native code is definitely pervasive in Android apps. While both benign and malicious code leverage native code, native invocations are substantially more common in malware (70% vs. 43%).
These results indicate that ignoring native code is a serious threat to validity in Android static code analysis.

11.4.2 RQ2: Bytecode-Native Invocation Extraction Comparison

Identifying native-to-bytecode and bytecode-to-native code invocations are key steps toward code unification. Our objective is to estimate to what extent the corresponding building block in JuCify is effective against a benchmark and against the state of the art.

Native to Bytecode: Fourtounis et al. [172] proposed an approach to detect exit invocations (i.e., native to bytecode invocations, c.f., Section 11.3.1) in native code via binary scanning. Their tool named Native-Scanner [182] has been developed as a plugin of a framework called DOOP [183]. Briefly, their tool scans binary files for string constants that match Java method names and Java VM type signatures and follows their propagation. In this way, they consider all matches as new entry points back to bytecode.

To compare our NativeDiscloser with Native-Scanner, we developed and released 16 benchmark apps. All these apps are executable Android apps and have been tested on a *Nexus* 5 phones with Android version 8.1.0. We design these apps to cover different situations, such as dynamic/static registration, chained invocations in native functions, parameter passing via structures and classes, string accessing via arrays and function returns, string obfuscation, etc. Table 11.2 presents the results obtained with both tools.

These results show that Native-Scanner misses a high number of exit invocations. We realized that Native-Scanner seems not to consider Android framework APIs (the tool misses the API invocations in all benchmark apps). Note that Native-Scanner is not specific to Android. This could explain why it does not consider Android APIs. The tool is also challenged by constant string obfuscation (app bm_9), which is also the case for NativeDiscloser. *bm14* implements fake method string constants in the native part. For this app, we can observe the over-estimation of Native-Scanner (i.e.,

a high number of false positives) while NativeDiscloser is unaffected. Finally, NativeDiscloser also failed with string constants passing via arrays and function returns as implemented in *bm15* and *bm16* respectively. Limitations of Angr could cause this in parsing pointers of pointers. Overall, compared to Native-Scanner, NativeDiscloser obtains significantly higher precision and recall.

Bytecode to native: We were unable to compare NativeDiscloser with Native-Scanner. Unlike our tool, Native-Scanner does not investigate (1) bytecode to native entry invocations and (2) the relations mapping between entry and exit invocations.

Note, however, that on our benchmark of 16 apps, NativeDiscloser yields 100% precision in finding both the entry invocations and the entry-to-exit relations and achieves a recall of 95.59% and 89.19%, respectively.

RQ2 answer: Compared to the state-of-the-art Native-Scanner, our NativeDiscloser extracts exit invocations with better precision and recall. Besides, it can provide extra information, including entry invocations (i.e., bytecode to native invocations) and relations with exit invocations, which is essential to generate comprehensive call graphs.

11.4.3 RQ3: Can JuCify boost static data flow analyzers?

In Section 11.3, we described how JuCify could approximate the behavior of native functions based on the information retrieved from signatures, parameters, return type, and bytecode methods called from native code via JNI. In this RQ, we check if this first-step approximation helps perform advanced static analyses such as data leak detection on a well-defined benchmark. We will assess the capability of JuCify on real-world applications in RQ4.

The benchmark we built for RQ3 contains 11 apps that we plan to integrate into DroidBench, an open test suite with hand-crafted Android apps to assess taint analyzers. Among these apps, 9 contain a flow going through the native world, and 2 do not contain any data flow (to detect potential false positives). Then, we apply the state-of-the-art Flowdroid taint-analysis engine before and after applying JuCify in our benchmark apps to show that Flowdroid can, likewise in [123], be *boosted*. Flowdroid detects paths from well-defined sources (e.g., `getDeviceId()`) and sinks (e.g., `sendTextMessage()`) methods in Android apps.

Benchmark construction: We identified 4 cases on which we built our 11 benchmark apps to assess the ability of tools to detect data leaks via native code:

a) Getter: Source in native code and sink in Java code

b) Leaker: Source in Java code and sink in native code

c) Proxy: Source in Java code and sink in Java code

d) Delegation: Source in native code and sink in native code

Note that "Source/Sink in native code" means that the call to a sensitive method is actually performed in native code, but the sensitive method is always a method from the Android framework accessed with JNI (e.g., calling with JNI the `getDeviceId()` from the native code). For each of these cases, at least one step happens in native. Figure 11.2 illustrates these four cases. The red dots represent tainted information from a source method, and the red arrows represent how this information flows in the program. The *Getter* use-case allows developers to get sensitive data from the native code to leak it in the Java world. The *Leaker* use-case allows developers to get sensitive data from the Java world to leak it in the native world. Regarding the *Proxy* use-case, the sensitive information is retrieved in the Java world, sent to the native world to "break" the flow,

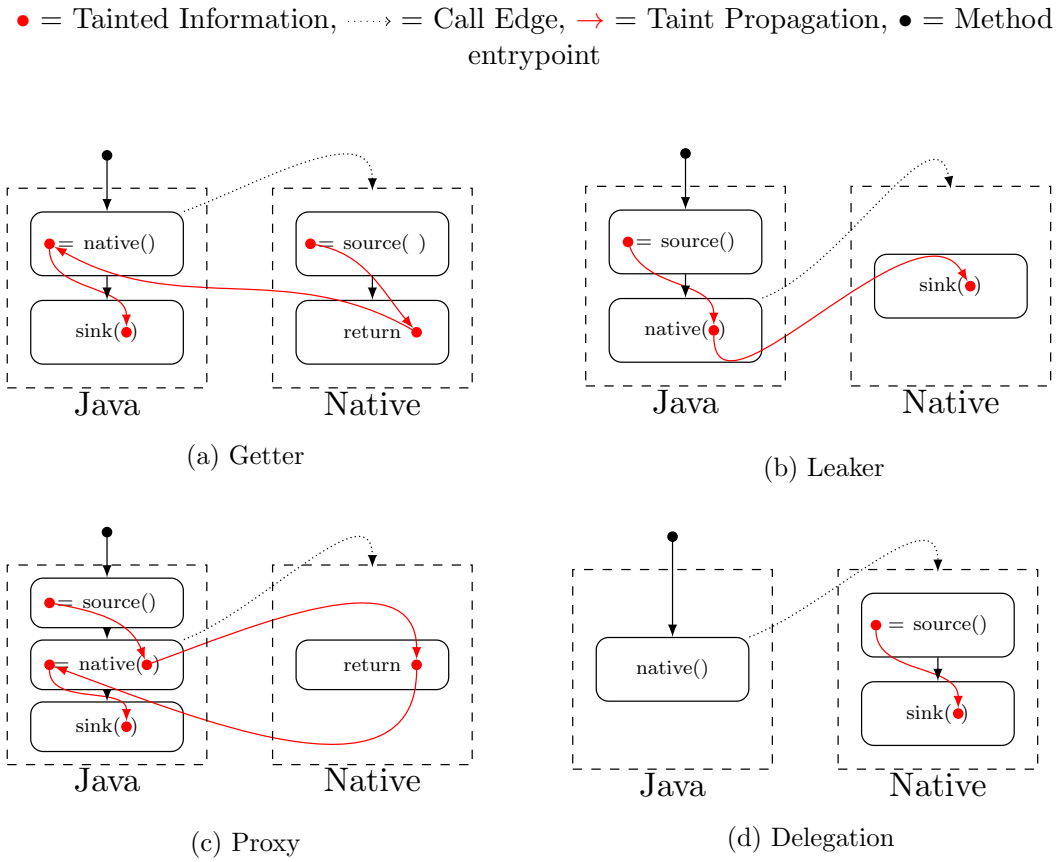


Figure 11.2: Four propagation scenarios through native code

and sent back to the Java world to be leaked. Concerning the *Delegation* use-case, a simple native function is called from the Java world, and the sensitive information is retrieved and leaked in the native world.

Our benchmark apps has been built, upon these four cases that we identified, to be representative of these cases, with combination of multiple cases.

Results: Table 11.3 provides the results of our experiments. FLOWDROID is clearly limited and not designed to handle native code. Therefore its inferior performances are not surprising. Indeed, FLOWDROID gets a precision and recall of 0% on this benchmark.

Nevertheless, we can see that after applying JUCIFY, FLOWDROID performance is significantly boosted. Indeed, it can detect all the leaks present in the benchmark, hence achieving a recall score of 100%. Regarding apps *getter_string* and *leaker_string*, FLOWDROID reports for both of them a false positive alarm leading to a precision of 82% on this benchmark. In these apps, a string is sent outside the apps, not sensitive data. This is easily explained by the fact that when JUCIFY reconstructs the native function's behavior, it uses opaque predicates to approximate what variable can be returned by the current function given its signature. Therefore, there is a path in which the sensitive data is considered, whereas it is not leaked.

RQ3 answer: JUCIFY is essential for boosting state-of-the-art static analyzers such as FLOWDROID to take into account native code. On our constructed benchmark, FLOWDROID, which failed to discover any leak, is now able to precisely identify leaks in a high number of samples (F1-score at 90%).

⊛ = true positive, ⋆ = false positive, ◯ = false negative

Test Case	Leak	Flowdroid	JuCify
getter_imei	•	◯	⊛
leaker_imei	•	◯	⊛
proxy_imei	•	◯	⊛
delegation_imei	•	◯	⊛
getter_string	○		⋆
leaker_string	○		⋆
proxy_double	•	◯	⊛
delegation_proxy	•	◯	⊛
getter_leaker	•	◯	⊛
getter_proxy_leaker	•	◯	⊛
getter_imei_deep	•	◯	⊛
Sum, Precision, Recall			
⊛, higher is better		0	9
⋆, lower is better		0	2
◯, lower is better		9	0
Precision $p = ⊛/(⊛+ ⋆)$		0%	82%
Recall $r = ⊛/(⊛+ ◯)$		0%	100%
F_1-score $= 2pr/(p+r)$		0%	90%

Table 11.3: Results of data leak detection through native code in bench apps. FLOWDROID column represents the results of running FLOWDROID alone. JUCIFY column represents the results of running FLOWDROID after applying JUCIFY

11.4.4 RQ4: JuCify in the wild

In this section, we evaluate JUCIFY in the wild from two points of view: ① a quantitative assessment in RQ4.a; and ② a qualitative assessment in RQ4.b.

RQ4.a: To what extent can JuCify augment apps' call graphs and reveal previously unreachable Java methods?

To assess to what extent call graphs are augmented by JUCIFY, we applied it to two sets of Android apps: ① 1000 benign apps; ② 1000 malware. Note that we only selected apps that contain at least one `.so` file. The results reported concern apps for which JUCIFY succeeded in making call graph changes. The reasons why there are apps without changes are related to the absence of bytecode-to-native links (i.e., for 559 goodware and 384 malware) and/or JUCIFY reaching the 1h-timeout (i.e., for 15 goodware and 51 malware).

Number of nodes and edges in call graphs: We first report the average number of nodes (i.e., the number of methods) and edges (i.e., the number of potential invocations) in the call graphs obtained before and after having applied JUCIFY.

The call graph augmentations brought by JUCIFY are visible in Table 11.4. Column *# apps* represents the number of apps for which JUCIFY made call graph changes, i.e., they did not reach the timeout and contained bytecode-native links. We notice that about half of the apps' call graphs are impacted by JUCIFY (426 and 565 for goodware and malware, respectively). We then notice that the number of nodes and edges added by JUCIFY is higher for goodware than for malware: 270 vs. 197 on average per app for nodes and 778 vs. 446 for edges. This shows that classical static analyzers that do not take into account the native code overlook a significant amount of nodes and edges in their call graph.

Number of binary functions in the augmented call graph: Newly added nodes

	# apps	Before JuCify		After JuCify		Difference	
		# Nodes	# Edges	# Nodes	# Edges	Added Nodes	Added Edges
Goodware	426	4515	18 287	4784	19 065	270 (+5.9%)	778 (+4.2%)
Malware	565	3056	14 266	3253	14 712	197 (+6.4%)	446 (+3.1%)

Table 11.4: Average numbers of nodes and edges before and after JuCify on 426 goodware and 565 malware

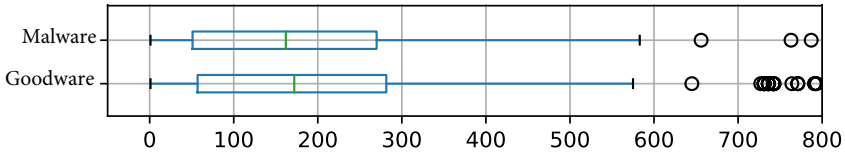


Figure 11.3: Distribution of the number of binary functions nodes in benign and malicious Android apps

can be explained by the binary functions (i.e., functions in the native code part) that are now considered in the unified call graph yielded by JuCify. Figure 11.3 details the distributions of the number of binary functions for both datasets. We notice that benign apps tend to have more added binary function nodes (median = 172, and mean = 269.7) in the call graph than malicious apps (median = 162, and mean = 197.2). Both distributions are significantly different, as confirmed by a Mann-Whitney-Wilcoxon (MWW) test [184] (significance level set at 0.05).

Number of bytecode-to-native call graph edges: Newly created edges can originate from native function invocations in bytecode methods (i.e., entry invocations). We compute the number of bytecode-to-native edges in the apps' call graph and detail their distributions over our datasets in Figure 11.4. The difference between malware and goodware is significant, with a median equal to 14 for malware and 8 for goodware. Overall, JuCify reveals a total of 6758 bytecode-to-native invocations in the malware dataset and 29 908 in the goodware dataset.

Number of native-to-bytecode call graph edges: Newly added edges can also originate from bytecode methods invoked in native functions (i.e., exit invocations with reflection-like mechanisms as explained in Section 11.2.1). The median number of edges is significantly low for both goodware and malware. Indeed, the median of native-to-bytecode edges is equal to 3 for both datasets. The distribution is available in Figure 11.5. Overall, JuCify reveals a total of 261 native-to-bytecode invocations in the entire goodware set and 4288 in the malware set. The conclusion that can be drawn from these results is the following:

the low numbers of native-to-bytecode edges in goodware show that benign apps make little use of reflection-like mechanisms to invoke Java methods from native code, compared to malware.

New previously unreachable bytecode methods: By considering native code, JuCify can reveal previously unreachable bytecode methods that are now reachable (because

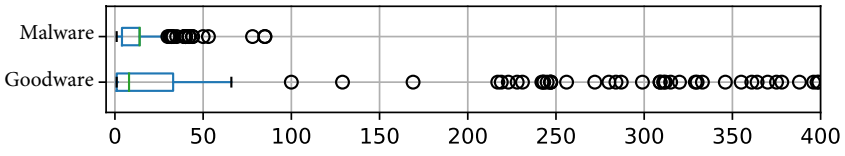


Figure 11.4: Distribution of the number of bytecode-to-native edges in benign and malicious Android apps

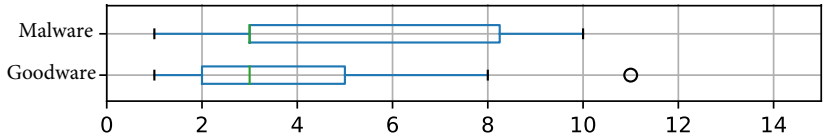


Figure 11.5: Distribution of the number of native-to-bytecode edges in benign and malicious Android apps

they are called from the native part). The number of previously unreachable bytecode methods is highly linked to the number of native-to-bytecode call graph edges discussed in the previous paragraph. However, a new edge from native to bytecode can simply end at a previously reachable node, which does not present an interest here. Indeed, newly reachable nodes are interesting since they allow static analyzers to not consider them as dead code anymore. In Section 11.4.3, we give a concrete example of the importance of this metric.

Overall, JuCify can reveal 34 previously unreachable bytecode methods in 18 benign apps (with a maximum of 5 for one given app). For malicious apps, JuCify reveals 122 previously unreachable bytecode methods called from native code in 54 apps. This accounts for 13% of native-to-bytecode invocation in goodware and 2.8% for malware. This suggests that in most cases, when Android app developers invoke bytecode methods from native code, it is to trigger bytecode methods that are already reachable from the bytecode. However, this shows that a non-negligible proportion of bytecode invocation from the native in goodware and malware are overlooked by classical static analyzers since they account for non-reachable nodes in the original bytecode call graph.

Goodware vs. Malware native/bytecode calls: To better understand the difference between goodware and malware, we inspected the native functions invoked from the bytecode and the bytecode methods invoked from the native code. Results indicate that in 82.7% of the cases, the native function `Java_mono_android_Runtime_register` is invoked from the bytecode in goodware. In fact, most of the top invoked native functions in goodware are from the `mono` framework, which is used by Xamarin [185]. The same method is, however, not found in the malware dataset. The top invoked native functions in malware is composed of different elements such as `Java_com_seleuco_mame4all_Emulator_setPadData`, `Java_com_shunpay210_sdk_CppAdapter210_pay`, or more suspicious functions: `Java_iqqxF_TZfff_ggior` and `Java_glrrx_efgnp_twCJN`.

From native to bytecode, we note some interesting insights: while benign apps invoke from the native code, in the majority of cases, bytecode methods like `Context.getPackageName` (14.2%), or `ThreadLocal.get` (8.2%), malicious apps invoke methods such as `TelephonyManager.getDeviceId` (2.4%), or `TelephonyManager.getSubscriberId` (4.3%) which can indicate suspicious behaviors.

Our results become more convincing by focusing on bytecode methods that were previously unreachable in call graphs and called from native code. While most of the bytecode methods that were previously unreachable and called in the native code in goodware are `Mono` framework methods, the situation is different in malware. Indeed, the most used bytecode methods in native code are dedicated to payment libraries (e.g., `com.shunpay208.sdk.ShunPay208`), and sensitive methods such as `getDeviceId`.

RQ4.a answer: JuCify helps to discover new paths in apps' behavior. It augments call graphs with about 5–6% new nodes in both benign and malware apps. Overall, apps tend to use much more bytecode-to-native invocations than native-to-bytecode. However, malware seems to use bytecode invocations from native code to perform suspicious activities.

RQ4.b: Can JuCify reveal previously unreachable sensitive data leaks that pass through native code in real-world apps?

With this RQ, our goal is to assess JuCify from a qualitative point of view. In particular, we check whether the call graphs augmented by JuCify with previously unseen nodes are relevant. To that end, we run JuCify and Flowdroid on real-world apps to check if Flowdroid can detect sensitive data leaks through the native code.
Experimental setup: To assess JuCify in the wild, we selected malicious applications since the intuition is that malicious apps tend to leak sensitive data more than goodware. Therefore, we randomly selected 1800 malicious apps (i.e., VirusTotal score > 20) from Androzoo [46] that contain .so files. Besides, to detect data leaks, we used the default sources and sinks provided by Flowdroid. For each of these 1800 apps, we set a 1-hour timeout (30 min for the symbolic execution and 30 min for Flowdroid).
Findings: Among the 1800 malicious apps, 1460 contained Java native methods declaration(s) in the code. In total, JuCify was able to augment the call graph of 1066 (i.e., 73%) of the 1460 apps that contain both .so files and Java native method declaration in bytecode. From these 1460 apps, Flowdroid revealed sensitive data leaks that take advantage of the native code in 14 apps. These 14 apps were manually checked and confirmed to contain sensitive data leaks that go through the native code. Note that this number is highly linked to the source and sink methods used.

In the following, we discuss two case studies where JuCify was able to reveal sensitive data leaks that pass through native code. Both Android apps were manually checked by the authors to confirm the presence of a leak detected by Flowdroid.

Getter-Scenario Case Study

In Figure 11.2a we illustrated an example of how malware developers can rely on native code to hide, from static analyzers, the *retrieval* of sensitive data from static analyzers. JuCify revealed an Android malware [2] implementing this specific behavior. JuCify reconstructed the `A()` native method from the `com.y` class as the following: "`java.lang.String Java_com_y_A(android.content.Context)`". In this native function, the IMEI number of the device is obtained via the JNI interface and returned as a result. This reconstructed method is called in method `b()` of class `com.cance.b.q` to store the IMEI number. The resulting IMEI number is then wrapped and transferred to a method to log it.

After examining the VirusTotal report of this app, we found that the flags raised by antiviruses refer to Trojan behavior and explicitly mention the retrieval of sensitive information from the device as well as the use of native code in the implementation of the malicious behavior. To some extent, this corroborates that JuCify contributed to uncovering a malicious behavior that is hidden through exploiting native-to-bytecode links (which state-of-the-art static analyzers could not be aware of).

Leaker-Scenario Case Study

In Figure 11.2b, we illustrated how app developers can rely on native code to hide the *leakage* of sensitive data. JuCify revealed an Android malware [3] with this behavior.

First, the IMEI number is obtained in the `getOperator()` method of the `com.umeng.adutils.AppConnect` class and stored in the `imei` field of the same class. Then, in the `processReplyMsg()` method of this class (which is triggered when an SMS is received), the IMEI number is wrapped in another string and sent to the native method

[2] SHA-256: 54DAFDF3635B18C0FD9F5CE89FE14C072D75AB4687B376FBADF370388574DC14
[3] SHA-256: A0B7BFBC272B462A2F59CC09ACC8B75114137CF7A2B391201C14C1A90EA7E369

"`stringFromJNI()`" as a parameter. JuCify's instrumentation engine constructed the following method from this native method: "`java.lang.String Java_com_umeng_adutils_SdkUtils_stringFromJNI(android.app.PendingIntent,java.lang.String,java.lang.String)`". This latter has been populated with the information given by the symbolic execution and revealed that the `sendTextMessage()` method from the `android.telephony.SmsManager` class is called with the value derived from the IMEI number as a parameter.

To summarize, a value derived from the IMEI number is sent out of the device using an SMS through the native code. Doing so, the leak would have remained undetected without JuCify.

As in the previous case study, we examined the VirusTotal report of this app. In their majority, antiviruses flag it as a Trojan app. Some reports even explicit tag the use of `getDeviceId()` and of native code for the malicious operations. Thus, with JuCify, we enabled an existing analyzer to uncover a leak being performed through native code.

RQ4.b answer: JuCify is effective for highlighting data flows across native code that were previously unseen. Indeed, its enhanced call graphs enable static analyzers to reveal sensitive data leaks within real-world Android apps.

11.5 Limitations

Our approach is a step towards realizing the ambition of full code unification for Android static analysis. Despite promising performances, our current prototype of JuCify presents a few limitations: First, our implementation relies on existing tools to extract native call graphs and mutual invocations between bytecode and native code. Limitations of these tools are therefore carried over to JuCify. Such limitations include the exponential analysis time for symbolic execution, the limitation in finding the boundaries of native functions, the unsoundness in app modeling with Flowdroid due to reflective calls [8], multi-threading [186], and dynamic loading [107].

Second, our prototype currently relies on symbolic execution, which is known to be non-scalable in the general case. Therefore, as described in Section 11.4.4, the call graph of some Android apps was not augmented due to the symbolic execution that did not return native-bytecode links and/or due to the timeout.

Third, a major limitation of JuCify lies in the fact that it does not yet reconstruct native functions' behavior with high precision. Indeed, as described in Section 11.3.2, for the native functions that represent Java native functions, JuCify considers a partial list of statements: it employs opaque predicates to guide static analyzers into considering every possible path during analyses. Moreover, JuCify overlooks native functions that are not explicitly targeted by JNI Java calls since it cannot approximate their behavior in the current implementation. As a result, JuCify cannot generate native functions' control flow graphs with Jimple statements covering the functions' full behavior. This limitation implies that if, for instance, a leak is performed by using Internet communication implemented "purely" in C (e.g., with a socket), then this leak would not be detected with Flowdroid even after JuCify processing. Also, during the reconstruction phase described in Section 11.3.2, the number of parameter combinations can explode in some cases where the number of parameters is important. This can lead to methods being extremely long that might not represent reality. We plan to address this limitation in future work, which we have already started by taking advantage of the symbolic execution results to account for parameters passed to functions as well as variables returned after function calls. The interested reader can refer to the project's repository for more information[4].

[4]https://github.com/JordanSamhi/JuCify

11.6 Threats to validity

Manual Checking. To check the correctness of the results, we manually checked a hundred Android apps. To do so, we relied on Java bytecode decompilers and native code decompilers such as Jadx [187] and RetDec [188]. Although native code manual checking is challenging, we were able to confirm that the native nodes added by JuCify matched the nodes from the native call graph constructed by NativeDiscloser. Regarding bytecode-to-native links, as the symbols were always available for the apps we checked (since native methods were statically registered), we were able to confirm the correctness of those links in the call graph generated by JuCify. We reverse-engineered these apps and were able to reach the same conclusions. Regarding native-to-bytecode links, the method names are represented as strings, which are not directly available in the native code. Therefore, we faced the challenge of checking if the symbolic execution yielded correct links. One way to verify would be to execute the code part to trigger the native code and ensure that the correct information is yielded by NativeDiscloser, but this is a challenge per se and it is out of the scope of this study. Therefore, we made the hypothesis that the symbolic execution yields correct results.

11.7 Related work

Static analysis of Android apps. Static analysis of Android apps is widely explored to assess app properties. Less than 10 years after the introduction of Android, a systematic literature review [104] has shown that over one hundred papers presented static approaches to analyze Android apps. The review highlights that Android apps' security vetting is one of the main concerns for analysts who assess properties such as sensitive data leak detection [5, 6, 123], or check for maliciousness [155, 189, 190]. Static approaches have also been implemented to identify functional and non-functional defects [9, 191] and towards fixing runtime crashes [192, 193]. Static analysis is also further leveraged to collect information in apps towards improving dynamic testing approaches [194, 195, 196, 197]. Given these fundamental usages of static analysis, it is essential to take into account all code that implements any part of the app behavior. Therefore, the fact that many analyses are reduced to focus on the bytecode (while leaving out native code within app packages) constitutes a severe threat to validity in many studies.

Binary analysis. Binary analysis techniques have been applied for different platforms, using static [177, 198, 199, 200], dynamic [168, 201, 202], hybrid [203, 204, 205] and machine-learning-based [206, 207, 208, 209] approaches. A recent work [172] tackles the challenging task of analyzing binaries by combining declarative static analysis (using Datalog declarative logic-based programming language) with reverse-engineering techniques to perform x-refs analysis in native libraries using Radare2 [210]. In the Android realm, analysis of binaries can be essential to cope with obfuscation [211].

Cross-language analysis. Several researchers have also acknowledged the presence of native code alongside bytecode in their analysis of Android apps. For instance, in 2016, Alam et al. [102] presented *DroidNative* which can perform Android malware detection considering both the bytecode and the native code. *NDroid* [167] and *TaintArt* [170] were proposed for dynamic taint analysis to track sensitive information flowing through JNI. *JN-SAF* [166] is also proposed as an inter-language static analysis framework to detect sensitive data leaks in Android apps, taking into account native code. All the aforementioned tools, however, are task-specific. They also, typically, perform their analyses separately for bytecode and native code, and later post-process and merge the outputs to present unified analysis results. In contrast, JuCify proposes to unify the representation before task-specific analyses. This enables other analyses to be built upon the output

of JuCify. For the experimental assessment of the JuCify representation for data flow analysis (RQ-5), we envisioned a comparison with JN-SAF. Unfortunately, two co-authors independently failed to run the tool.

Overall, there are various approaches and studies [171, 212, 165, 169] in the literature that investigate the possibility of analyzing apps by account for the different language-specific artifacts in the package. Although the approaches described are promising for cross-language analysis, they do not generally offer a practical framework to unify the representation of both the bytecode and the native code into a single model that standard static analysis pipelines can leverage. Our prototype JuCify does bring such a unified model and targets the Jimple intermediate representation, which is the default internal representation of Soot. Therefore, by pushing in this research direction, we expect to provide the community with a readily usable framework, which will allow them to (re)perform their analyses on whole code in Android apps.

11.8 Summary

We contribute to the ambitious research agenda of unifying bytecode and native code to support comprehensive static analysis of Android apps. We presented JuCify as a significant step towards this unification: it generates a native call graph that is merged with the bytecode call graph based on links retrieved via symbolic execution. In this model (i.e., the unified call graph), we are able to heuristically populate specific native functions with Jimple statements. The Jimple intermediate representation was selected to readily support existing static analyzers based on the Soot framework.

We first empirically showed that JuCify significantly improves Android apps call graphs, which are augmented (to include native code nodes) and enhanced (to reveal previously unreachable methods). Then, we showed that JuCify holds its promise in supporting state-of-the-art analyzers such as Flowdroid in enhancing their taint tracking analysis. Finally, we discuss how JuCify can reveal sensitive data leaks that pass through the native code in real-world Android apps, which were previously undetectable.

Chapter 12

Resolving Conditional Implicit Calls for Comprehensive Analysis of Android Apps

In this chapter, we push further our will to expose unreachable code from static analyzers in Android apps. Indeed, we investigate conditional implicit calls that impede both static and dynamic analyses. Static analysis may overlook the code's possible behaviors or over-approximate possible behaviors leading to false positive alerts. Dynamic analysis may fail to provide inputs satisfying the constraints triggering the delegation of execution control. We developed and evaluated ARCHER*, a tool that resolves conditional implicit calls and extracts the constraints that trigger the delegation of execution control. Our empirical evaluations show that* ARCHER *allows both static and dynamic analyzers to cover more Android apps' code.*

This chapter is based on our work submitted in the following research paper:

- **Jordan Samhi**, Ye Qiu, René Just, Michael D. Ernst, Tegawendé F. Bissyandé, and Jacques Klein. Archer: Resolving Conditional Implicit Calls for Comprehensive Analysis of Android Apps. *In peer review for the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).*

Contents

12.1 Overview

Android dominates the mobile market, both in terms of the number of supported devices [213] and the number of apps [214]. Therefore, security and privacy in Android are important for practitioners and researchers. Previous research investigates ① static analysis [2, 3, 4, 215, 6, 7], ② dynamic analysis [12, 13, 14], and ③ hybrid analysis [31, 29]. Despite this research, malware developers can evade existing analyzers by exploiting implicit call capabilities in the Android framework. Implicit calls trigger the execution of methods without a direct call in the code under analysis. For instance, Android life-cycle methods, such as `Activity.onCreate()`, are never called in apps' code.

This work's focus is two-fold: ① improving static analysis and ② aiding dynamic analysis of Android apps. Static analysis tools typically rely on control flow graphs [25] and call graphs [216]. In the case of data flow analysis, an algorithm would typically start at an entry point method and propagate data flow values along the control flow graph. When a method call is encountered, the analysis continues to other methods, using the potential target methods computed by the call graph. Call graph construction algorithms such as CHA [57], RTA [217], VTA [59], Andersen [218], Steensgaard [219], SPARK [50], etc. cannot natively resolve implicit calls. Therefore, static analysis tools that rely on these algorithms will overlook parts of the code, which need to be visited to ensure comprehensive analysis.

In addition to implicit calls, the Android framework enables developers to constrain the execution of the code targeted by implicit calls. For instance, the `JobService.onStartJob()` method, which is called implicitly using method `JobScheduler.schedule()`, can be constrained to be executed only if the device battery is charging, or if the network to which the device is connected is the cellular one, etc. This is achieved by using several classes and methods of the Android framework such as: `JobInfo.Builder.setRequiresCharging(true)`, or `JobInfo.Builder.setRequiredNetworkType(4)`. These constraints challenge dynamic analyzers, which would not execute the code targeted by **C**onditional **I**mplicit calls (***CI** calls*) if the constraints are not met at run time.

In contrast with CI calls, non-conditional implicit calls have already been well-explored by the research community. Several approaches have been proposed to improve static analyzers by connecting different types of implicit calls to potential target methods. However, these works focus on specific implicit calls such as life-cycle methods [5], reflection [18, 8], callbacks [133], or inter-component communication (ICC) [6]. Since these mechanisms have already been studied in the literature, they are out of the scope of this work.

Our work complements and extends prior work by modeling Conditional Implicit Calls, i.e., implicit calls that trigger methods under certain criteria, e.g., at a specific date, which are currently overlooked in the literature. Indeed, attackers can use these techniques to evade both static and dynamic analysis. Moreover, hiding code is common for malware developers [220]. Hence, if malware developers use CI calls, they can escape static and dynamic analyzers and enter the Google Play [220]. Note that existing techniques cannot be applied to resolve the CI calls studied in this work since it requires additional processing to compute the potential targets using specific techniques (cf. Section 12.4).

In this work, to cope with the limitation of the state of the art regarding CI calls, which require specific processing to be resolved statically, we propose a novel approach, ARCHER, to: ① statically resolve CI calls in Android apps for boosting current static analyzers; and ② extract the constraints needed to be met to execute them to aid dynamic analyzers. This work makes the following contributions:

- We systematically searched the Android framework for implicit calls that can be triggered under circumstances.
- We provide an empirical analysis of Android apps showing that CI calls are prevalent.

```
1  public class MainActivity extends Activity {
2    public static String secret;
3    @Override
4    protected void onCreate(Bundle b) {
5      super.onCreate(b);
6      secret = getSecret();
7      NetworkType nt = NetworkType.CONNECTED;
8      Constraints cs = new Constraints.Builder()
9        .setRequiredNetworkType(nt)
10       .setRequiresCharging(true).build();
11     Class c = MyWorker.class;
12     OneTimeWorkRequest wr = new OneTimeWorkRequest.Builder(c)
13       .setConstraints(cs).build();
14     WorkManager.getInstance(this).enqueue(wr);
15   }
16   private String getSecret() { /* code */ }
17 }
18 public class MyWorker extends Worker {
19   public MyWorker(Context c, WorkerParameters w){
20     super(c, w);
21   }
22   @Override
23   public Result doWork() {
24     String secret = MainActivity.secret;
25     leak(secret);
26     return null;
27   }
28 }
```

Listing 12.1: Code illustrating how a CI call with constraints can be triggered within an Android app. The call to method `enqueue()` on line 15 triggers method `doWork()` on line 24 if the criteria set in the `Constraints` object are met.

- We propose ARCHER, a novel approach to resolve CI calls, improve Android apps' call graph, and extract the constraints implicit calls need to meet to be executed.
- We release a new benchmark to assess CI-call-aware tools.
- We evaluated ARCHER: ① it augments Android apps' call graphs; ② it outperforms existing static analyzers; ③ it extracts the criteria needed to be met to trigger implicit calls with high precision; and ④ it allows dynamic analyzers to improve their code coverage.
- Our prototype's implementation and our experimental scripts and data are publicly available at:

`https://github.com/JordanSamhi/Archer`

12.2 Motivation & Background

Classical call graph construction algorithms are challenged by CI calls, affecting the comprehensiveness of static analyzers. Dynamic analyzers are challenged by CI calls since they might not meet constraints needs to cover parts of the code.

12.2.1 A Motivating Example

Listing 12.1 shows an example of a data leak in an Android app using a CI call. We explain why this data leak is not detected by state-of-the-art static data leak detectors such as FLOWDROID, and why it might not be detected by dynamic analyzers.

On line 6, a sensitive datum is stored into field `MainActivity.secret`. The call to `getSecret()` is explicit, i.e., the call site is directly connected to the `getSecret()` method implementation. Lines 7–10 build a `Constraints` object requiring that the network is connected, and the device is charging. Then, lines 11–13 show the construction of a `OneTimeWorkRequest` object that points to class `MyWorker` and is fed with the `constraints` object. On line 14, a `WorkManager` instance calls the `enqueue()` method with the work request as a parameter.

After the call to `enqueue()`, method `MyWorker.doWork()` (lines 23–26) will be executed if the conditions set by the constraints are met. When analyzing an Android app's code, resolving the relationship between `enqueue()` and `doWork()` is challenging since the implicit mechanism works at the Android framework level. This poses problem for both static and dynamic analyses: ① static analyzers that are not CI-call-aware will miss the data leak since the relation between `enqueue()` and `doWork()` is not modeled; ② Dynamic analyzers might miss the data leak since several conditions have to be met to trigger its execution.

To overcome these problems, this work aims at: ① augmenting call graphs with these CI calls; and ② extracting the potential conditions (set by constraints) that need to be met to execute the code for dynamic analyzers in order to improve their code coverage.

12.2.2 Definitions

This section introduces concepts that are used in the work.

Explicit call of a method m is a method call directly referring to m in the code under analysis. For instance, on line 6 of Listing 12.1, the call to method `getSecret()` directly refers to method `getSecret()`.

Implicit call of a method m triggers the execution of m without a direct call to m in the code under analysis. For instance, in Listing 12.1, the `doWork()` method is executed, after the call to `enqueue()`, though there is no direct call to method `doWork()` in the app.

Conditional Implicit call (CI call) is an implicit call triggered under certain conditions. For instance, in Listing 12.1, the `doWork()` method is executed if two conditions are satisfied: ① the device is connected to the Internet (line 9); and ② the device is in charge (line 10).

Executor. We refer to classes and methods that trigger CI calls as *executor* classes and *executor* methods. In line 15 of Listing 12.1, class `WorkManager` is an executor class and method `enqueue()` is an executor method.

Executee. After calling an executor method, an *executee* method will be executed; its class is an *executee* class. In Listing 12.1, `MyWorker` is an executee class. On line 24, `doWork()` is an executee method.

Helper. We refer to any classes or methods, other than executors and executees, that participate in the implicit method call mechanism as *helper* classes and *helper* methods. An example of a helper class in Listing 12.1 is the `Constraints` class on line 8. Examples of helper methods are `setRequiredNetworkType()` and `setRequiresCharging()` (on lines 9–10).

Data flow analysis. A data flow analysis is a static analysis technique used to compute properties (e.g., sets of possible types of a variable) at different points in a program.

Data flow problem. A data flow problem is characterized by: ① a domain D of data flow values that need to be computed at program points; ② an initial data flow value from which a data flow analysis will start the analysis; ③ a set of transfer functions for each program point p such as $f_p : D \to D$; and ④ an operator to combine information from multiple predecessors (e.g., $\cup$).

IFDS framework. Solving a data flow problem can be achieved by the IFDS framework. The IFDS framework reduces context-sensitive inter-procedural analysis problems

to propagate information (i.e., dataflow values) in programs into graph reachability problems. The IFDS framework relies on an *exploded super graph* in which nodes represents dataflow values at given program statements, and edges represent transfer function behavior. In this framework, transfer functions can be of four kinds:

1. Normal edges: propagate dataflow values from a statement that does not contain a procedure call to its successors.
2. Call edges: propagate dataflow values from call sites to callee methods.
3. Return edges: propagate dataflow values from return statements to call site receivers.
4. Call-to-return edges: propagate intra-procedural data flow values from a statement containing a method call to its successors

(The interested reader can refer to Figure 2 in [221] which depicts these different edges.) If a given node $n = (s, d)$, such as s is a program statement and d a dataflow value, is reachable from the initial abstract dataflow value 0, it means the dataflow value d holds at statement s.

12.3 Collecting methods enabling conditional implicit calls

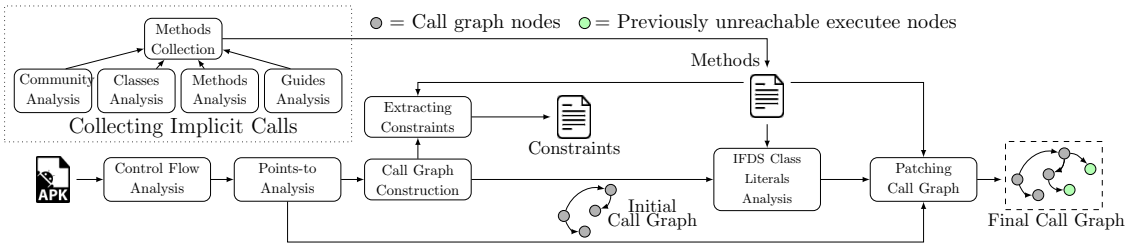


Figure 12.1: Overview of our approach to resolving implicit calls and constraints

To the best of our knowledge, the literature does not provide a comprehensive list of CI call mechanisms in the Android ecosystem. A major contribution of our work is to yield such a list for the community. This section overviews our methodology for this identification.

Concretely, we are interested in implicit calls for which the execution can be constrained. We collected classes that provide a CI call mechanism from several sources.

Community. We asked on Stack Overflow [222] for Android mechanisms that trigger code under given circumstances. We received one answer that pointed us to the `WorkManager` class. We performed a snowball analysis to find similar mechanisms: ① we carefully read the webpage [223] associated with the `WorkManager` class to collect the classes involved in this particular mechanism, then ② we followed any hypertext link that could lead us to similar mechanisms and applied step ① on any new page found. This revealed 12 classes: 2 executor classes, 2 executee classes, and 8 helper classes.

Classes analysis. We looked for *job-like* mechanisms. We manually examined the 493 classes whose name contains one of the following strings in the Android source code (API 30):

```
"trigger", "schedul", "criter", "execute", "delay", "work", "job", "time"
```

We found 27 classes related to triggering code execution: 19 executor classes, 6 executee classes, and 2 helper classes. 4 (i.e., 1 executor, 1 executee, 2 helpers) overlap with those

from the "community" point above, bringing the total number of classes of interest to 35 (i.e., $12 + 27 - 4$).

Methods analysis. We looked for ways to trigger code under time-related circumstances, by manually examining the 545 methods in the Android framework that have a parameter whose name contains one of the following strings:

```
"second", "minute", "milli", "hour", "delay"
```

This analysis yielded a single new executor class, which brings the number of classes of interest to 36 (i.e., $35 + 1$).

Documentation analysis. We read every webpage in the online Android guides [224], which explain how to implement certain mechanisms. We found 5 pages [225, 226, 227, 228, 229] that discuss implicit call mechanisms, but all refer to classes and methods we already collected.

In the end, our collection yielded 21 executor classes, 7 executee classes, and 8 helper classes.

Methods collection. So far, we have described how we collected Android classes that enable CI calls. However, an analysis tool needs to work at the *method* level to improve its models and to discover implicitly executed code. To do so, we carefully studied the documentation of the classes that we collected and how they can be used to trigger CI calls. Concretely, we searched for methods that could fall into one of our categories, i.e., executor, executee, or helper methods. We found 60 executor methods, 6 executee methods (two executee classes share the same executee method), and 25 helper methods. Based on the documentation, we manually produced a list of pairs of executor-to-executee methods to improve static analyzers and a list of helper methods that will support our analysis to extract conditions under which CI calls might occur. Table 12.1 shows the different constraints that executor classes can use to trigger executees thanks to helper methods. Most executors allow setting time-related constraints, i.e., to set a delay before executing an executee, or a period of execution of an executee. Five executors allow to set the execution of an executee as persistent across device reboots, i.e., the executee will run even after the device reboots. Only one, the `SoundTriggerDetector`, can trigger some code according to the sound detected by the device. Six executors allow to set constraints depending on device states: ① the status of the network, i.e., connected or not; ② the type of the network, cellular or Wi-Fi; ③ the battery level, i.e., low or not; ④ the charging status, i.e., in charge or not; ⑤ the idle status, i.e., currently idle or not; and ⑥ the storage level, i.e., low or not.

We provide a language in ARCHER to easily incorporate new CI call mechanisms that we might have missed through our systematic study (cf. Section 12.6). Our annotated lists of classes and methods are publicly available in our project's reproduction artifacts.

12.4 Approach

This section presents ARCHER, a tool that aims at: ① resolving CI calls to improve static analyzers; and ② extracting the conditions that need to be met to execute CI calls to aid dynamic analyzers. It has been shown in the literature [104] that most Android static analyzers are built on top of the Soot static analysis framework [24], hence we implemented our approach in the form of a tool, ARCHER, on top of Soot. This section introduces the overall conceptual approach implemented in ARCHER.

12.4.1 Resolving conditional implicit calls

Call graph construction computes, for each call site, the potential targets or procedure implementations that may be invoked at run time. For object-oriented dispatch, this

	Constraints									
Class	Delay	Periodic	Persistent	Sound	NS	NT	BL	CS	IS	SL
Timer	✓	✓								
SoundTriggerDetector				✓						
WorkManager	✓	✓			✓		✓	✓	✓	✓
JobSchedulerShellCommand	✓	✓	✓		✓	✓	✓	✓	✓	✓
JobScheduler	✓	✓	✓		✓	✓	✓	✓	✓	✓
JobSchedulerImpl	✓	✓	✓		✓	✓	✓	✓	✓	✓
JobSchedulerService	✓	✓	✓		✓	✓	✓	✓	✓	✓
JobServiceContext	✓	✓	✓		✓	✓	✓	✓	✓	✓
CompletableFuture	✓	✓								
ExecutorCompletionService										
Executor										
HandlerExecutor										
SynchronousExecutor										
RepeatableExecutor	✓	✓								
RepeatableExecutorImpl	✓	✓								
DelayableExecutor	✓	✓								
ExecutorImpl	✓	✓								
ExecutorService	✓	✓								
ScheduledExecutorService	✓	✓								
ScheduledThreadPoolExecutor	✓	✓								
AbstractExecutorService										

Table 12.1: Different constraints that can be set to executor classes to trigger CI calls. (NS = Network Status, NT = Network Type, BL = Battery Level, CS = Charging Status, IS = Idle Status, SL = Storage Level)

determines which overriding method implementations might be executed. This can be done via (for example) a type-based analysis or a points-to analysis [219].

In some cases, this same points-to information can indicate, for a given statement, which methods could executed via a CI call. For example, the `CompletableFuture.runAsync()` method takes a `Runnable` object as a parameter. Hence thanks to the points-to set and our executor–executee mapping, our analysis would infer that the `run()` method would be triggered.

In other cases, the points-to information is not adequate. Consider Listing 12.1 and the call to method `enqueue()` on line 15 with variable `wr` as the argument. The points-to set for variable `wr` is of no use; rather, one of the `wr`'s fields indicates the executee. In particular, `wr` is a `OneTimeWorkRequest` object that was created on line 12 with a class literal `MyWorker.class`. In other words, the `OneTimeWorkRequest` object wraps a reference to class `MyWorker`. Standard static analyzers overlook the connection between the `enqueue()` method and the implementation of the `doWork()` method (lines 24–27), hence the leak cannot be detected. The same mechanism is used by Android's `JobScheduler` class which relies on `JobInfo` objects wrapping `ComponentName` objects which in turn wrap class literals. The following explains how Archer resolves the potential target of such wrapper objects.

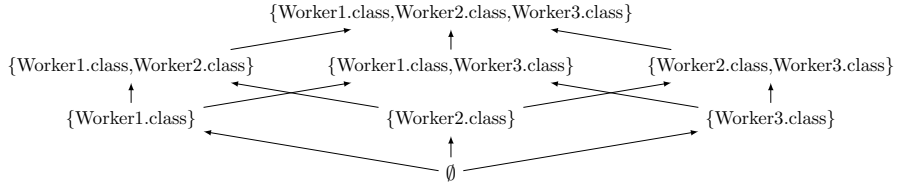


Figure 12.2: Powerset lattice of three class literals

IFDS Class Literals Analysis. The problem we want to solve is the propagation of class literal information to wrapper objects that are used as arguments to executor methods. This problem is not a trivial data flow propagation analysis. Algorithm 12.1 gives our solution. We illustrate its operation on the `WorkManager` example depicted in Listing

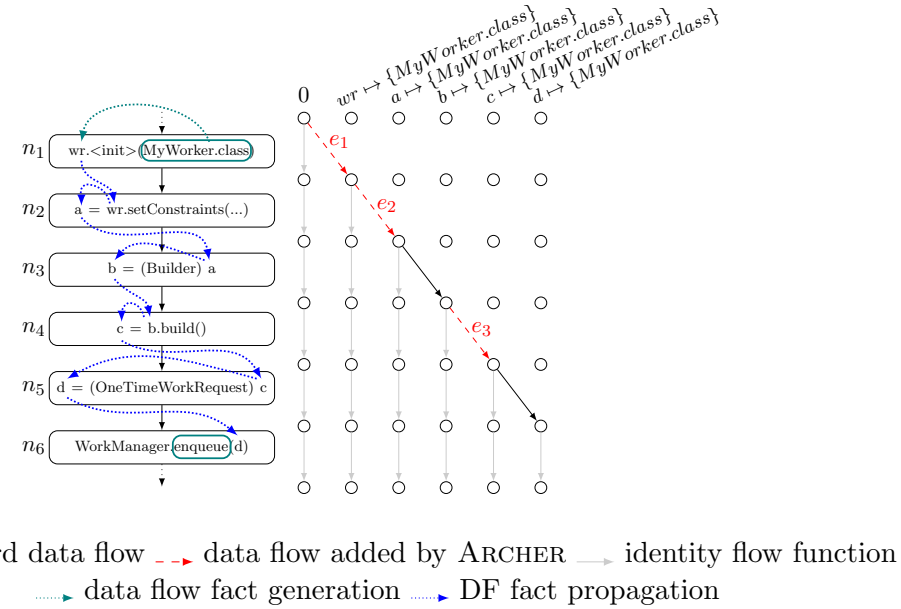


Figure 12.3: How Archer handles call-to-return transfer function within the IFDS framework to generate new data flow values to propagate class literals. For simplicity, the figure omits call and return transfer functions.

12.1.

Archer propagates class literals using the Inter-procedural Finite Distributive Subset (IFDS) [230] framework. Let C be the set of class literals in the code of a given app. The abstract domain is the lattice $D = (\mathcal{P}(C), \subseteq)$ formed by $\mathcal{P}(C)$ the powerset of C. Figure 12.2 illustrates the abstract domain of an app with three class literals. To assign an abstract value to each variable of the program, we use a mapping $S = V \mapsto D$, where V represents the set of variables in the app. Hence, a data flow value represented by: $v \mapsto X$ such as $v \in V$ the set of program variables and $X \in \mathcal{P}(C)$ the powerset of C which is the set of class literals in a program, means that v holds a reference to class literals in X.

Contrary to standard analyses which implement call-to-return transfer functions as the identity function to propagate data flow values intra-procedurally after a procedure call, Archer handles these transfer functions differently. The algorithm on our project's repository main page shows how ARCHER handles the generation and propagation of data flow values after method calls.

For instance, consider Figure 12.3 which depicts part of the code of Listing 12.1, transformed into Jimple [45], the internal Soot [24] representation. For ease of explanation, we simplified the Jimple code generated. Let us start with node n_1 in which the `<init>(MyWorker.class)` constructor is called on variable `wr`. With a standard transfer function, no data flow value would be generated for variable `wr` (other than it already holds) after the call to its constructor. Moreover, n_1 is not an assignment statement. Hence `wr` does not receive any value returned by the call to its constructor. Therefore the return transfer function would not propagate any data flow value to it. Also, a standard *call-to-return* transfer function would apply the *identity* function and intra-procedurally propagate any data flow value reachable before n_1. However, in our case, we want to know what possible class literal variable `wr` might transitively refer to, possibly through fields, after node n_1. Therefore, our analysis generates data flow value $wr \mapsto \{MyWorker.class\}$ after calling `<init>(MyWorker.class)` on variable `wr` (edge e_1 on Figure 12.3, and lines 34–38 on our algorithm in the project's repository). The same reasoning is applied for nodes n_2 and n_4 in which we propagate the data flow values of

Algorithm 12.1 Transfer functions for class literals data flow analysis. An edge $\langle s_0, d_0\rangle \rightarrow \langle n, d\rangle$ means that, according to the analysis, data flow value d holds at statement n if and only if data flow value d_0 holds at statement s_0. `workList` temporarily stores edges that serve to propagate data flow values. `pathEdges` stores the edges from the initial node to reachable nodes in the exploded super graph. `callToBase` is a set of methods taking class literals as parameter that we manually vetted (see Section 12.3) which generate new dataflow values for caller objects (e.g., $a.f(c)$ would generate a new dataflow value $a \mapsto \{c\}$). `callToReceiver` is a set of methods that we manually vetted which propagate dataflow values held by caller object to a potential receiver (e.g., $a = b.f()$ would propagate any dataflow value held by b to a).

```
1: workList := {⟨S0, 0⟩ → ⟨S0, 0⟩}
2: pathEdges := {⟨S0, 0⟩ → ⟨S0, 0⟩}
3: callToBase := initializeCallToBase()
4: callToReceiver := initializeCallToReceiver()
5: procedure PROPAGATE(n1, d1, n2, d2)
6:     for s ∈ succ(n2) do
7:         edge := ⟨n1, d1⟩ → ⟨s, d2⟩
8:         if edge ∉ pathEdges then
9:             Insert edge in pathEdges
10:            Insert edge in workList
11:        end if
12:    end for
13: end procedure
14: procedure IFDS( )
15:    while workList ≠ ∅ do
16:        Select an edge ⟨nx, dx⟩ → ⟨ny, dy⟩ from workList
17:        switch ny do
18:            case ny is an assignment a = b
19:                // e.g., n3 and n5 in Figure 12.3
20:                if b is a class literal then
21:                    Propagate(nx, dx, ny, a ↦ {b})
22:                else
23:                    if dy is b ↦ X then
24:                        propagate(nx, dx, ny, a ↦ X)
25:                    end if
26:                end if
27:            case ny is an assignment a = b.f()
28:                if f ∈ callToReceiver then
29:                    // e.g., n2 and n4 in Figure 12.3
30:                    if dy is b ↦ X then
31:                        propagate(nx, dx, ny, a ↦ X)
32:                    end if
33:                end if
34:            case ny is a call statement a.f(c)
35:                if f ∈ callToBase then
36:                    // e.g., n1 in Figure 12.3
37:                    if c is a class literal then
38:                        Propagate(nx, dx, ny, a ↦ {c})
39:                    else
40:                        if dy is c ↦ X then
41:                            propagate(nx, dx, ny, a ↦ X)
42:                        end if
43:                    end if
44:                end if
45:            Propagate(nx, dx, ny, dy)
46:    end while
47: end procedure
48: // for brevity we omit details of standard call transfer functions such as formal/actual parameters
    mapping or summary function computation and return transfer functions.
```

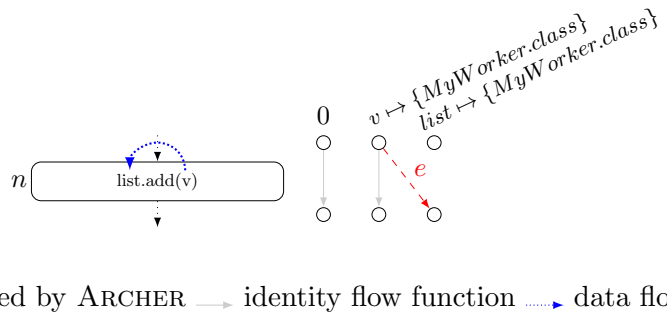


Figure 12.4: How Archer handles call-to-return transfer function within the IFDS framework to generate new data flow values to propagate data flow facts in collection-like objects. For simplicity, the figure omits call and return transfer functions.

variables `wr` and b to variables a and c respectively generating the following data flow values: $a \mapsto \{MyWorker.class\}$, and $c \mapsto \{MyWorker.class\}$ (cf. lines 27–33 on our algorithm in our project's repository). Without doing so, at node n_6, the analysis would never know that variable d is potentially referring to class `MyWorker`. This allows our analysis to know that the argument given to the `enqueue()` method is of type `MyWork`, hence it can correctly connect method `enqueue()` to method `doWork()` of class `MyWorker` thanks to our currated mapping of executor–executee. The entire data flow propagation is depicted in Figure 12.3 with dotted arrows showing how the data flow facts are generated (from a class literal in the rounded rectangle in node n_1) and propagated to an executor (i.e., rounded rectangle in node n_6).

Handling Collections We have previously seen how ARCHER can determine the possible targets of executor methods. However, for several methods collected (see Section 12.3), it is not enough since they take, as a parameter, a collection of objects. For instance, as explained so far, our strategy would work for method `WorkManager.enqueue(WorkRequest wr)`, but not for method `WorkManager.enqueue(List⟨WorkRequest⟩ wrs)`. Indeed, we need to add an extra step to propagate the data flow values to collection-like objects. To do this, we propagate the data flow values held by parameters of function calls that allow populating collection-like objects such as lists, sets, etc. Figure 12.4 depicts this process where in our analysis, variable `list` is mapped to any data flow values held by variable v. This process allows ARCHER to know that the collection-like argument given to the `WorkManager.enqueue(List⟨WorkRequest⟩ wrs)` method holds a reference to class `MyWork`, hence it can correctly connect method `WorkManager.enqueue(List⟨WorkRequest⟩ wrs)` to method `doWork()` of class `MyWorker`.

Patching Call Graph. After collecting the potential targets of *executor* method calls, we rely on our list of pairs of executor-to-executee methods previously constructed (see Section 12.3). Basically, for every potential target of an executor method call, we retrieve the corresponding executee method (e.g., in the example of Listing 12.1, if `enqueue` is the executor method, then `doWork()` is the executee method) and patch the call graph with an edge from the executor method to the executee method of the potential target class.

12.4.2 Extracting constraints

With CI calls, a developer can set execution constraints using method calls (e.g., `setRequiresCharging(true)` on line 10 in Listing 12.1). Consequently, to collect the different constraints that are defined to trigger an *executee* method, we perform an interprocedural data flow analysis using the IFDS framework, and the following specific configuration : Let M be the set of methods that allow setting a constraint on the execution of an *executee*.

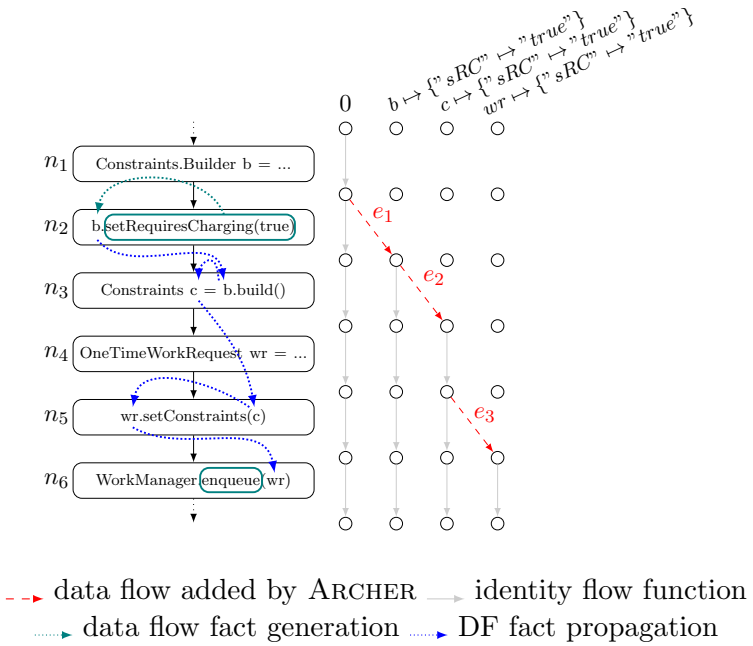


Figure 12.5: How Archer handles call-to-return transfer function within the IFDS framework to generate new data flow values for constraints propagation. For simplicity, the figure omits call and return transfer functions. (sRC = setRequiresCharging).

Let `Val` be the set of values used to set the actual constraints (e.g., value `true` on line 10 of Listing 12.1). Let C be the Cartesian product of M and `Val`. Then, the abstract domain is the lattice $D = (\mathcal{P}(C), \subseteq)$ formed by $\mathcal{P}(C)$ the powerset of C. We need to assign each program variable an abstract value to propagate the abstract values until a call to an *executor* is found. Therefore, we rely on S, a map lattice defined as follows: $S = V \mapsto D$, where V represents the set of variables in the app. As before, each element of S represents an abstract state of the information propagated. Hence, a data flow value is represented by: $v \mapsto X$ such as $v \in V$ the set of program variables and $X \in \mathcal{P}(C)$ the powerset of C which is the Cartesian product set of program methods and program values that allow setting a constraint to execute a CI call, means that v holds the information that m_1 was called with value `true`.

Likewise resolving the CI calls, ARCHER handles *call-to-return* transfer functions in a non-standard manner. Indeed, the data flow values need to be propagated to any object that will contribute to the creation of potential class target wrappers. For instance, consider Figure 12.5 which depicts parts of the code of Listing 12.1 simplified. To know that certain conditions are set for the `OneTimeWorkRequest` object passed as argument to the `enqueue()` method (node n_6), ARCHER needs to propagate the data flow values to the `Constraint.Builder` object (node n_2 for which ARCHER generates edge e_1), then to the `Constraint` object (node n_3 for which ARCHER generates edge e_2), and eventually to the `OneTimeWorkRequest` object (node n_5 for which ARCHER generates edge e_3). Eventually, our analysis can know the conditions that need to be satisfied for an executor method to be executed.

12.5 Evaluation

This section answers the following research questions:

RQ1: To what extent are CI calls prevalent in Android apps?

RQ2: How well does ARCHER perform?

RQ2.a: To what extent can ARCHER augment Android apps call graphs?

RQ2.b: To what extent can ARCHER extract precise conditions under which a CI call might be executed?

RQ3: How does ARCHER compare against existing state-of-the-art prototypes?

Datasets: To evaluate ARCHER, we relied on two datasets:

- **benchmark_dataset:** we developed 16 benchmark apps that use several executor/executee/helper classes and methods, as well as constraints to trigger executees under certain criteria. We inserted data leaks in 12 apps. We left 4 apps without leaks to assess false positives. We intend to contribute these apps to DroidBench [144].
- **real_world_dataset:** we collected two datasets from AndroZoo [46] with the following criteria: ① a dataset of 3000 benign apps (an app is considered as benign if none of the 60+ antiviruses from VirusTotal [55] has flagged it); and ② a dataset of 3000 malicious apps (an app is considered as malicious if at least 5 of the 60+ antiviruses from VirusTotal [55] has flagged it). We considered only apps with code size smaller than 10MB and dex file date of 2020 or later (i.e., recent apps compatible with Android API level 30). The average size and the standard deviation for malware is respectively 6.95 and 2.05, for goodware it is respectively 6.05 and 2.45.

12.5.1 RQ1: How prevalent are CI calls in real-world apps?

This section presents a quantitative analysis of CI calls usage. To perform this study, we used our **real_world_dataset**. We present the results from the point of view of both benign and malicious apps to show their differences.

	Executor	Executee	Helper
Benign apps	1239 (41.3%)	2670 (89%)	359 (12%)
Malicious apps	200 (6.7%)	343 (11.4%)	114 (3.8%)

Table 12.2: Number of apps that use CI calls.

Results Table 12.2 shows the number of apps that use at least one executor, executee, and helper classes and methods, respectively. A significantly higher proportion of goodware relies on CI calls. Executees are more used than executors and helpers in benign and malicious apps. The explanation is twofold: ① an executor can trigger several executees; and ② the executees studied include class `Runnable`, which is widespread in Android apps to trigger `Threads`. Table 12.3 shows the occurrences of different executees. The `Runnable` class is, by far, the most used executee in both benign and malicious apps.

Although CI calls are used in more benign apps than malicious apps, Figure 12.6 shows that the median number of executors per app (with at least one executor) is 30.5 in malicious apps while it is only 4 in benign apps. Regarding executees, the median number per app (with at least one executee) is 34 in malicious apps, while it is 15 in benign apps. However, Figure 12.6 shows that helpers are more numerous in benign apps with a median of 6, while in malicious apps, the median is 2 (for apps with at least one helper).

RQ1 answer: CI calls are pervasive in Android apps. Although these mechanisms are more common in benign apps than in malicious apps, the median number of executors and executees is higher in malware. The results indicate that a substantial portion of code might be overlooked by static analyzers that are not CI-call-aware.

	# in Benign apps	# in Malicious apps
Total	134 636	45031
`Runnable`	121 148	39 696
`Callable`	10 357	5048
`TimerTask`	2838	231
`JobService`	234	54
`Worker`	56	2
`RunnableScheduledFuture`	3	0

Table 12.3: Number of occurences of executees identified thanks to our classes and methods collection (see Section 12.3)

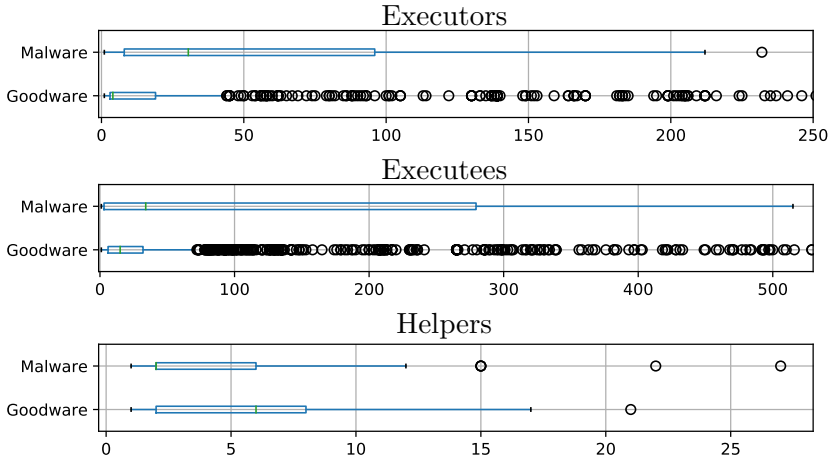


Figure 12.6: Distribution of the number of executors, executees, and helpers in our dataset of benign and malicious apps

12.5.2 RQ2: What is the performance of Archer?

In this section, we evaluate Archer's performance in: ① augmenting Android apps call graphs ; and ② extracting precise information about the conditions under which a CI call might be executed.

RQ2.a: Augmenting Android apps' call graphs

Archer is evaluated on real-world apps to assess to what extent it can augment apps' call graphs. We used our **real_world_dataset** and retained apps having calls to executor methods, i.e., 1239 goodware and 200 malware (see Table 12.2).

New call graph nodes and edges: We measured the number of new nodes and new edges in call graphs after applying Archer on our dataset (before Archer means that apps are loaded in Soot without our approach). Note that a call graph node represents a method and a call graph edge represents the calling relationship between two methods. Figure 12.7 shows the distribution of the number of new call graph nodes and edges brought by Archer. We notice that more nodes and more edges (in absolute numbers) tend to be added per app in malicious apps' call graphs than in benign apps. Regarding malware, the median number of new nodes and new edges are both 64, while for goodware, the medians are 10 and 11, respectively. Table 12.4 reports the average numbers of nodes and edges in the call graph before and after Archer. We notice that for both malware and goodware, there are more edges than nodes on average since a node can be the target of multiple edges.

These numbers show that static analyzers that do not model CI calls will miss a high number of methods to analyze per app. Indeed, in both goodware and malware, more than

	Before Archer		After Archer		Difference	
	# Nodes	# Edges	# Nodes	# Edges	Added Nodes	Added Edges
Goodware	3915.9	17 186.8	4008.8	17 346.6	92.9 (+2.37%)	159.8 (+0.93%)
Malware	5899.9	22 776	6023.1	23 019.7	123.2 (+2.09%)	243.7 (+1.07%)

Table 12.4: Average number of nodes and edges

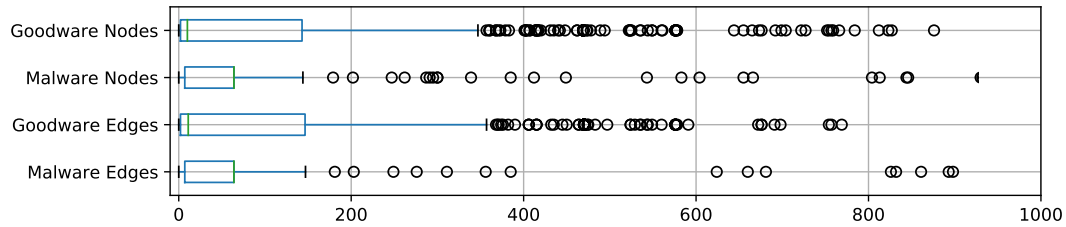


Figure 12.7: Distribution of new call graph nodes and edges

2% of nodes are added, i.e., overlooked by non-CI-call-aware tools. In the case of malware detection, this can cause malicious apps to enter the Google Play since the malicious code could be hidden in a CI call and then overlooked by static analyzers.

Extra code reachable: We report the number of previously unreachable statements and are made reachable thanks to ARCHER. Note that in our study, a statement is represented by a Jimple statement [45] since our implementation relies on Soot [24]. Figure 12.8 illustrates the distribution of the number of extra statements reachable thanks to ARCHER.

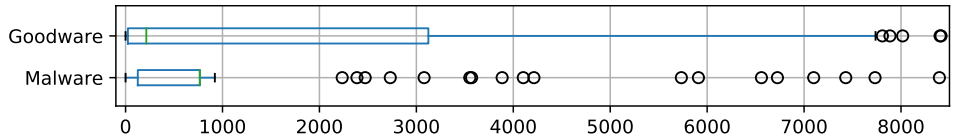


Figure 12.8: Distribution of extra Jimple statements

The average number of extra statements reachable for benign apps is 1741.2, and the median is 214. The average number of extra statements reachable for malware is 2295.1, and the median is 766. This shows again that standard and non-CI-call-aware static analyzers are missing a substantial part of the code when analyzing Android apps. Worse, it shows that a substantial part of code in *malicious apps* is overlooked, which is dangerous if malicious developers heavily rely on CI calls.

Analysis time overhead: To model an Android app, ARCHER (which is built on top of Soot) needs extra steps to resolve CI calls compared to other tools relying on Soot (e.g., FLOWDROID). We compute the time overhead brought by ARCHER to model Android apps. The time overhead is computed by measuring the time taken by ARCHER to resolve CI calls compared to the time Soot takes to load an app. Figure 12.9 shows the overhead introduced by ARCHER in both benign and malicious apps. On average, ARCHER brings an overhead of 38.1% (i.e., +27s on average) in malicious apps and 27.6% (i.e., +24s on average) in benign apps. The median for malware is 35.6% and 24% for benign apps. These results suggest that ARCHER brings an overhead that is not insignificant to compute an Android app model that can be used to implement any downstream analysis. Overall, the analysis time is higher in malicious apps than in benign apps.

Manual analysis To assess whether the edges that have been added are correct, we manually analyzed ARCHER's output.

Among the 210 853 call graph edges (48 008 + 162 845), from an executor to an executee, added by ARCHER in both goodware and malware, we randomly chose 96 edges (confidence level of 95% and confidence interval of ± 10%). For every edge (*executor* →

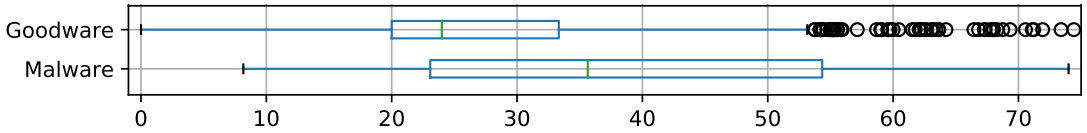


Figure 12.9: Distribution of the time overhead introduced by ARCHER to resolve CI calls (in %). The time overhead is the extra time taken by ARCHER to compute an Android app model (i.e., resolving CI calls).

executee), we followed this process: ① download app from AndroZoo [46]; ② decompile the app using Jadx [187]; ③ manually analyze the method in which `executor` is called to check the argument used to trigger it; and ④ if the argument used to trigger the executor is related to `executee`, we mark it as correct, incorrect otherwise.

For instance, consider Listing 12.1. If ARCHER yields an edge $WorkManager.enqueue() \rightarrow MyWorker.doWork()$, we manually analyze method `MainActivity.onCreate()` (since `enqueue()` is called in `MainActivity.onCreate()`) and check what are the argument(s) used to trigger it. In this case, the argument is `wr`, which holds a reference to class `MyWorker` (lines 12–13). Hence the edge is marked as correct.

Overall, we found that 81/96 of the edges analyzed are correct (precision 84%). All of the 15 incorrect edges were added by ARCHER using the results of the points-to analysis (see Section 12.4.1) used to construct the initial call graph. Hence, ARCHER suffers from the over-approximation of the points-to algorithm used, i.e., SPARK [50] in this case, to infer potential targets of some CI calls. This matter is further discussed in Section 12.6. For transparency, we open-source our annotated results in the project's replication artifacts.

RQ2.a answer: ARCHER discovers previously unreachable methods (> 2% on average) for inter-procedural data flow analyses. ARCHER's precision is 84%, and the imprecision is due to the SPARK points-to analysis used in our experiments.

RQ2.b: constraints for executing CI calls

This section evaluates ARCHER's ability to yield precise constraints needed to be met to trigger CI calls. To do so, we executed ARCHER on: ① our **benchmark_dataset**; ② our **real_world_dataset** for which we retained apps having calls to executor methods, i.e., 1239 goodware and 200 malware (see Table 12.2).

Benchmark apps On our benchmark apps, ARCHER achieves a perfect score. The results are visible in Table 12.5. Indeed, for all apps without constraint, ARCHER does not report any constraint. For all apps with constraints, ARCHER reports all the constraints with the exact parameters. For instance, in sample `WorkManager_enqueueUniqueWork`, the CI call is triggered if and only if: ① the device is not in an idle state; ② the device is connected to the Internet; and ③ the device needs to be in charge. ARCHER reveals these constraints without which a dynamic analyzer not fulfilling these criteria would overlook the code in the executee and not cover it.

Real-world apps First, we report the number of apps that make use of constraints when using executors: 596/1239 (48.1%) for goodware and 98/200 (49%) for malware. A total of 1395 constraints are found in the goodware dataset, and 208 in the malware dataset.

Second, we report the distribution of the number of constraints found per app. Results are depicted in Figure 12.10. For both goodware and malware, the median number of constraints found per app is 0. This is in line with the fact that ARCHER found constraints in almost half of goodware and malware. When constraints are used, we notice that both goodware and malware use several constraints per app, even more than 3 in 25% of the cases in goodware and more than 2 in 25% of the cases in malware.

⊛ = true positive, ◯ = true negative
• = constraint, ∘ = no constraint

Test Case	# Constraints	# Detection
WorkManager_enqueue	••	⊛⊛
WorkManager_enqueueUniqueWork	•••	⊛ ⊛ ⊛
WorkManager_enqueue1	•	⊛
TimerTask_schedule	•	⊛
JobScheduler_schedule	•••	⊛ ⊛ ⊛
JobScheduler_schedule1	∘	◯
CompletableFuture_runAsync	∘	◯
CompletableFuture_thenRun	∘	◯
CompletableFuture_runAsync1	∘	◯
ExecutorCompletionService_submit	∘	◯
ScheduledThreadPoolExecutor_scheduledAtFixedRate	•••	⊛ ⊛ ⊛
ScheduledThreadPoolExecutor_invokeAll	∘	◯
ScheduledThreadPoolExecutor_scheduleWithFixedDelay	•••	⊛ ⊛ ⊛
ExecutorService_submit	∘	◯
Poolexecutor_scheduleWithFixedDelay_enqueue	•••	⊛ ⊛ ⊛
SynchronousExecutor_execute	∘	◯

Table 12.5: Results of constraints detection on our benchmark

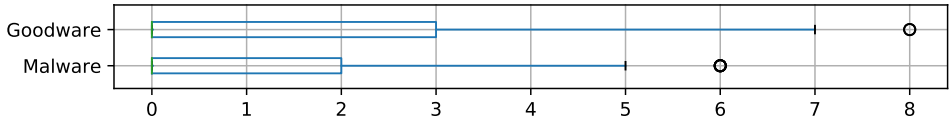


Figure 12.10: Distribution of the number of constraints per app

According to our experiments, when constraints are used, they are mostly time-related. Indeed, 50% of constraints in goodware require triggering the executee after a certain delay and 71% in malware. Other constraints involved in goodware are: run executee periodically (13%), requires that the device is connected to a specific network type (9%), requires that the device needs to be in charge (9%), requires that the device is in an idle state (9%), executee is persistent (4%), the latest constraint is the time unit extracted to set time-related constraints (6%). Other constraints involved in malware are: requires that the device needs to be in charge (9%), requires that the device is connected to a specific network type (9%), executee is persistent (2%), requires that the device is in an idle state (1%), run executee periodically (1%), the latest constraint is the time unit extracted to set time-related constraints (7%).

The conclusions that can be drawn are: ① dynamic analyzers might not cover part of the code in almost half of the apps if conditions to trigger CI calls are not met; ② several constraints are found per app that condition the triggering of executees which can complicate the tasks of dynamic analyzers to cover the code triggered; ③ the difference of more than 20% between goodware and malware for time-related criteria highlight the importance of extracting them to fine-tune dynamic analyzers and allow covering more code to, e.g., spot malicious code.

Manual analysis Since static analyzers often suffer from false positive results, we manually analyzed ARCHER's results. To do so, a sample of 91 constraints found by ARCHER was randomly selected out of 1603 (1395 + 208) with a confidence level of 95% and confidence interval of ± 10%. For each sample, the strategy is the following: ① the corresponding app is downloaded from AndroZoo; ② the app is decompiled using Jadx; ③ the method in which the executor is called is analyzed; and ④ if the constraints found by ARCHER (e.g., trigger after 10 seconds) correspond to the code decompiled, we mark it as correct, incorrect

otherwise. For instance, consider Listing 12.1, if ARCHER would yield the following constraints: ① the device needs to be connected to a network; and ② the network needs to be in charge, to execute the `Worker.doWork()` method, we would manually analyze method `MainActivity.onCreate()` and check what are the constraints set to trigger the executee. In this case, both methods `setRequiredNetworkType(NetworkType.CONNECTED)` and `setRequiresCharging(true)` are used to set the constraints, hence the constraints yielded by ARCHER are marked as correct in this case.

Overall, we found that the constraints found by ARCHER were correct in 86 cases out of 91 (94.5%). However, for the remaining 5 apps, the results were not marked as incorrect per se. Indeed, for these 5 cases, the types of the constraints were correct (e.g., a specific time before execution). However, the values used to set the constraints were not statically computed since they did not hold constant values. For transparency, we release our annotated results in the project's repository.

Archer's benefit to dynamic analysis To validate whether the constraints yielded by ARCHER can indeed aid dynamic analyzers, we rely on the ACVTool [231] dynamic analyzer, which produces the proportion of code covered when an app is executed. Our empirical setup is the following: we use ACVTool to monitor apps' execution in an emulator with ① the constraints extracted by ARCHER satisfied, and ② the constraints not satisfied. In both cases, we use the same inputs generated with Google's Monkey [232]. The results reported represent the code coverage proportion of the main app package (i.e., not library code).

We first report the results on our **benchmark_dataset**. We retained apps with constraints in the code (see Table 12.5) and code executed in the executee. Results are available in Table 12.6. We notice that for all apps, if the constraints yielded by ARCHER are not satisfied, the code coverage is greatly reduced, up to 28.6% of code not covered. After verification of the reports, we confirm that executee methods were not triggered when the conditions were not set accordingly.

	C satisfied	**C** not satisfied
WorkManager_enqueue	80%	65%
WorkManager_enqueueUniqueWork	83.3%	70.8%
WorkManager_enqueue1	78.4%	59.4%
TimerTask_schedule	70.4%	55.5%
JobScheduler_schedule	77.5%	62.5%
ScheduledThreadPoolExecutor_scheduledAtFixedRate	73.3%	60%
Poolexecutor_scheduleWithFixedDelay_enqueue	73.5%	44.9%

Table 12.6: Code coverage with and without proper constraints set in the execution environment. (**C** = Constraints)

We present a case study for a real-world app (i.e., org.wifi.analyzer.network.boost) of our **real_world_dataset** that triggers the execution of a `TimerTask` executee directly after the app is launched and under the following constraint: needs to wait 10 seconds before triggering the executee. When monitoring the app with ACVTool for more than 10 seconds, the anonymous executee class `MainActivity$2` (which extends `TimerTask`) obtains a coverage of 100%. If the app is run for less than 10 seconds, i.e., the constraint is not respected, `MainActivity$2` obtains a coverage of 0%.

This indicates that without proper inputs and configuration that can be proposed by ARCHER, dynamic analyzers would overlook the code in constrained executees and cover less code. The reports generated by ACVTool, as well as our scripts to assess the benefits for dynamic analyzers, are available in our project's repository.

⊛ = true positive, ⋆ = false positive, ◯ = false negative,
• = leak, ∘ = no leak

Test Case	Leak	FlowDroid	IccTA	RAICC	Amandroid	DroidSafe	DroidRA	Archer
WorkManager_enqueue	•	◯	◯	◯	◯	◯	◯	⊛
WorkManager_enqueueUniqueWork	•	◯	◯	◯	◯	◯	◯	⊛
WorkManager_enqueue1	∘							
TimerTask_schedule	•	⊛	◯	⊛	◯	◯	⊛	⊛
JobScheduler_schedule	•	⊛	◯	⊛	◯	◯	⊛	⊛
JobScheduler_schedule1	∘	⋆		⋆			⋆	
CompletableFuture_runAsync	•	◯	◯	◯	◯	◯	◯	⊛
CompletableFuture_thenRun	•	◯	◯	◯	◯	◯	◯	⊛
CompletableFuture_runAsync1	∘							
ExecutorCompletionService_submit	•	◯	◯	◯	◯	◯	◯	⊛
STPE_scheduledAtFixedRate	•	◯	◯	◯	◯	◯	◯	⊛
STPE_invokeAll	•	◯	◯	◯	◯	◯	◯	⊛
STPE_scheduleWithFixedDelay	∘							⋆
ExecutorService_submit	•	◯	◯	◯	◯	◯	◯	⊛
PE_scheduleWithFixedDelay_enqueue	•	◯	◯	◯	◯	◯	◯	⊛
SynchronousExecutor_execute	•	◯	◯	◯	◯	◯	◯	◯
Sum, Precision, Recall								
Count of ⊛, higher is better		2	0	2	0	0	2	11
Count of ⋆, lower is better		1	0	1	0	0	1	1
Count of ◯, lower is better		10	12	10	12	12	10	1
Precision $p = ⊛/(⊛+ ⋆)$		67%	0%	67%	0%	0%	67%	92%
Recall $r = ⊛/(⊛+ ◯)$		16.7%	0%	16.7%	0%	0%	16.7%	92%
F_1-score $= 2pr/(p+r)$		26.7%	0%	26.7%	0%	0%	26.7%	92%

Table 12.7: Data leak detection on **benchmark_dataset**

RQ2.b answer: Half of the apps using CI calls rely on constraints to trigger the code, which challenges dynamic analyzers to cover parts of the code. Archer can precisely report constraints used to trigger CI calls in 94.5% of the cases. Archer's constraint extraction aids dynamic analyzers in covering more code.

12.5.3 RQ3: Comparison with state of the art

To assess how Archer's effectiveness in resolving conditional implicit flows, we compare the precision and recall of *available and usable* state-of-the-art data flow analyzers that take into account implicit flows on our **benchmark_dataset**. We considered the following tools: ① FlowDroid [5]; ② IccTA [6]; ③ RAICC [123]; ④ Amandroid [11]; ⑤ DroidSafe [135]; and ⑥ DroidRA [8].

Results Table 12.7 summarizes the results of this experiment. The six state-of-the-art tools studied perform poorly on *this* dataset since they are not designed to handle CI calls. FlowDroid, RAICC and DroidRA (since RAICC and DroidRA rely on FlowDroid for the data flow analysis) did detect three leaks for these reasons: ① in sample `TimerTask_schedule` because FlowDroid implements a hard-coded heuristic for this particular case (see [215], lines 564–647); ② in sample `JobScheduler_schedule` because the `JobService` class used (an executee) is an Android `Service` component, and FlowDroid considers its methods inherited by Android framework classes (i.e., `JobService` is), as potential callbacks as an over-approximation. Consequently, the code in method `onStartJob()` of class `JobService` is covered by FlowDroid; and ③ in sample `JobScheduler_schedule1` for the same reasons. However, this third reported leak is a false positive result. Indeed, method `onStartJob()` is triggered using methods such as `schedule()`, but in this sample, it is not the case. Hence there is no leak at run time. Since FlowDroid models method `onStartJob()` as potentially called, it reports a false positive.

IccTA, Amandroid, and DroidSafe yield a zero score on *this* dataset. Indeed, none of these tools can reach and analyze the code written in *executees* since they cannot resolve the conditional implicit flows studied in this work.

ARCHER's performance outperforms state-of-the-art prototypes with precision, recall, and f_1 score of 92%. Indeed, ARCHER allows uncovering previously non-reachable parts of the code in which the leaks have been put for this dataset. Hence more code is analyzed. We note: ① that for sample `JobScheduler_schedule1`, no false positive is detected since we do not apply FLOWDROID's over-approximation for the `onStartJob()` method; ② ARCHER issues a false positive for sample `ScheduledThreadPoolExecutor_scheduleWithFixedDelay` in which there is no leak at run time since the constraints under which the executee should be executed will never be met. Indeed we have set the time to trigger the executee in the past. This shows a limitation of ARCHER that does not statically check if the constraints are realizable to resolve a CI call, hence it over-approximates; and ③ ARCHER issues a false negative for sample `SynchronousExecutor_execute` in which the executee is triggered using reflection, ARCHER does not handle reflection. We provide the scripts to execute third-party tools in our project's repository for reproduction purposes.

RQ3 answer: ARCHER outperforms state-of-the-art static data flow analyzers for detecting conditional implicit flows in Android apps. On a well-defined benchmark, ARCHER achieves a 92% F_1 score, whereas FLOWDROID, RAICC and DROIDRA yield a 16.7% score, and AMANDROID, IccTA and DROIDSAFE achieve a 0% F_1 score.

12.6 Limitations and threats to validity

We manually analyzed the Android documentation to collect ways to perform CI calls in the Android framework. Although we followed a systematic approach, we might have missed some mechanisms. We mitigate this threat to validity in two ways: ① we share all our artifacts to the community for further checking and exploration; and ② if we missed some CI call mechanisms, new ones can easily be integrated into ARCHER.

ARCHER over-approximates certain CI calls behavior. In the case that such a mechanism does not allow to execute a particular executee at run time since some constraints will never be met (e.g., triggering an executee in the past, see Section 12.5.3), ARCHER still connects the executor and potential executees involved. Future work could check if the constraint might be feasible at run time.

ARCHER relies on points-to analysis results to infer the potential target of executor methods. Therefore, it shares the limitations of the algorithm used to compute the points-to set of variables, i.e., the over-approximation of the targets.

12.7 Related Work

This section discusses other research on resolving specific types of implicit control flow. It is complementary to our work in that ours is the first to catalog and analyze general Android framework mechanisms for implicit calls (triggering of code under specific circumstances). None of the previous work handles these sorts of triggers. A tool should utilize both previous techniques and also our new contributions.

Inter-Component Communication Android apps are made of *components* communicating through inter-component communication (ICC) [233]. Components are triggered with ICC methods provided by the Android framework (e.g., `startActivity()`, `startService()`) [5] that trigger *lifecycle methods* execution. Each component implements its own lifecycle methods (e.g., `onCreate()`, `onBind()`, etc.). ICC methods trigger lifecycle methods which represent implicit calls.

A large body of work tries to resolve target components of ICC communication. IccTA [6] infers `Intent` potential targets using IC3 [139]. AMANDROID [11] infers possible target components by generating a component-wise data flow graph and component-level

data dependence graph. AMANDROID's engine then relies on a summary table that models ICC channels. DROIDSAFE [135] relies on string and class designator analysis to infer potential target components, then DROIDSAFE modifies ICC method calls into explicit lifecycle method calls. RAICC [123] was proposed to resolve ICC links when atypical ICC methods are used (e.g., `AlarmManager.set()`). Atypical methods usually rely on `PendingIntent` or `IntentSender` objects wrapping component targets. After resolving potential targets with new IC3 rules, RAICC instruments the app and adds well-documented ICC method calls. ICCBot [234] was recently released as a new tool to infer the component transition (i.e., ICC) that are connected via Android's fragments. ICCBot performs a context-sensitive and inter-procedural analysis to precisely model data carried by ICC objects (e.g., Intents). Chen et al. [235, 236] developed an approach to construct an Activity Transition Graph (ATG) to build Android apps' storyboards. To do so, the authors rely on ICC-related information used to trigger new Activities.

Callbacks Wu et al. [237] proposed a callback-aware approach to detect resource leaks in Android apps. The authors focus on two types of callback methods: ① system-triggered callbacks; and ② user-triggered callbacks. The former represents, in their study, lifecycle methods and resource classes' callback methods (e.g., `onPause()`) while the latter represents callbacks triggered by user interaction with the GUI. Similarly, Yang et al. [238] study lifecycle and user-driven callbacks. Their approach relies on a GUI model by generating a callback control flow graph. The authors then extract possible sequences of user GUI events derived from valid paths in the GUI model.

EdgeMiner [133] is a tool that automatically retrieves callback capabilities in the Android framework. The authors focus on the registration mechanisms that allow passing procedures as parameters. A classical example in the Android ecosystem is the `setOnClickListener()` method that triggers the `onClick()` method. Yang et al. [239] proposed an approach based on the effect of GUI-related callbacks to construct a model of the behavior of Android apps' user interfaces.

Reflection Reflection permits *introspection* at execution time. For instance, `method.invoke(obj)` triggers the execution of the method represented by `method` on `obj`, but the method's name does not appear in the source code at that location.

DROIDRA [8, 132] is an instrumentation-based analysis to boost Android apps. The authors resolve reflective calls using the COAL [139] solver to infer reflection targets. Eventually, they instrument the app, and for each reflective call resolved, a corresponding explicit call is added in the app.

Besides `Intent` objects resolution, Barros et al. [18] proposed to resolve Java reflection calls targets. Their solution is two-fold: ① a reflection type system tracking and inferring potential names of classes and methods; ② a reflection solver estimating method signatures that can be invoked.

Although Java reflection resolution is an important topic to improve static analyzers w.r.t. implicit flow, our work does not target reflection. We focus on Android framework mechanisms provided to developers allowing conditional implicit flow.

Other Pan et al. [10] explored five techniques to perform asynchronous tasks. They implement AsyncChecker and conduct a qualitative analysis to check for misuse in Android apps.

You et al. [240] study the possible implicit flow that can arise in Android's Dalvik bytecode. They develop a control-transfer-oriented analysis in a formal structured semantic model. They show that several Dalvik instructions are responsible of implicit flow: ① if, ② switch, ③ throw, etc.

Fengguo et al. [241] explore how Android malware use task scheduling to trigger malicious code. For instance, they discovered that malware relies on recurring tasks with `Thread` objects to receive commands from external servers.

Contrary to these approaches, we do not aim to provide qualitative information about mechanisms implementation. We propose to improve state-of-the-art static analyzers with previously overlooked links between method calls that can be used to trigger code under specific circumstances.

12.8 Summary

ARCHER: ① improves static analyzers by revealing previously hidden code due to CI calls, and ② aids dynamic analyzers by revealing the conditions under which CI call targets might occur. Experiments show its effectiveness: ① CI calls studied are well spread in Android apps. ② ARCHER improves apps' call graphs by augmenting them with new edges that reveal previously unreachable code. ③ ARCHER can reveal previously undetected data leaks ④ ARCHER provides precise information regarding the conditions under which CI calls occur, aiding dynamic analyses to cover more code.

Chapter 13

Part II Conclusion

In Part II, we presented our work to improve the comprehensiveness of Android apps' static analysis. The Android ecosystem is designed to make apps heavily communicate with the Android framework. Therefore, as described in Chapter 1, since the framework acts as a black box to static analyzers, there are discontinuities that make static analysis challenging and make parts of apps' code statically unreachable. Our work shows that static analysis is an unavoidable technique toward continuing digging into the Android framework to find mechanisms that allow implicit calls in order to improve the comprehensiveness of Android apps' static analysis.

More specifically, we have made the following contributions: ① we have proposed RAICC, an approach to reveal and resolve atypical inter-component communication in Android apps. Our work shows many ICC links are overlooked by state-of-the-art tools. RAICC participates in improving the comprehensiveness of Android apps' static analysis. ② we have proposed JuCify, an ambitious approach to unify the bytecode and the native code in Android apps to support comprehensive static analysis. Our experiments have shown that JuCify support and improve state-of-the-art static analyzers for static data flow analysis in native code, usually overlooked by standard analyzers. ③ we have proposed Archer, a new static analysis approach to reveal and resolve condition implicit calls in Android apps. Experimental results show that conditional implicit calls are well spread, Archer can reach previously unreachable code for static data flow analysis, and that Archer provides additional information to dynamic analyzers regarding the conditions to trigger conditional implicit calls. Therefore, dynamic analyzers can cover more code with adequate inputs and environment settings.

In general, even though our work contributed to improving the comprehensiveness of static analysis for Android apps, analyses cannot be *holistic* yet. Our work is a step toward to ambition to make static analysis of Android apps comprehensive. However, several mechanisms hinder the current state of static analysis from being comprehensive. We discuss some of them in Chapter 14.

Chapter 14

Future Work

In this chapter, we present a research agenda to move further and thoroughly our contribution to the next level towards improving Android static analysis comprehensiveness.

Contents

The following three topics have been identified as requiring further investigation: ① hidden code in apps, ② the increasing prevalence of multiple programming languages in Android apps, and ③ the complexities of the Android framework. These topics are discussed in more detail below.

14.1 Hidden code

Android apps are nothing more than a collection of files packaged together. Files containing code are well identified, e.g., Dalvik bytecode, native libraries, etc. Android apps contain many resource files which, as described in Chapter 2, can represent, e.g., images, videos, sound files, etc., which apps need in order to deliver appropriate service to end users. However, it has been shown that malware developers often rely on techniques to hide malicious code in files such as images [242] (e.g., using steganography).

Existing techniques to statically analyze Android apps only consider conventional files where code can be found in order to analyze it. Therefore, if code is hidden in obscure places in the APK file, it is overlooked, which highly restrains the scope from searching for interesting properties, e.g., malicious code. Hence, accounting for hidden code in files in Android files is a promising future research direction that is in line with our past research.

14.2 Languages

The multi-language trend is real in Android apps development. Android apps are usually developed using the Java or Kotlin languages. The Android framework provides built-in functionalities to allow developers to use the C and C++ languages thanks to the Native Development Kit. Also, many frameworks have been developed to allow developers to use other programming languages in Android apps. For instance, Flutter [243] allows the use of the DART programming language; Ionic [244] allows developers to use technologies such as HTML, CSS, or SASS; React Native [245] provides a complete framework to integrate JavaScript in apps; Cordova [246] allows to incorporate web-like languages in apps. Numerous additional languages can be used to build apps, e.g., C#, Python, LUA, etc.

The current state of the art in Android apps' static analysis focuses on the conventional files embedding Dalvik bytecode or even native code (i.e., C and C++) recently. Therefore, static analysis of recent apps misses many apps' code and behavior, especially if apps are only written with a single technology, e.g., a web app. Thus, future work in Android apps' static analysis must ride the wave of the multi-language trend and propose novel approaches to account for different languages in Android apps.

14.3 Android framework

We have seen, in this manuscript, that the Android framework provides several mechanisms to trigger implicit calls in Android apps. In this case, the Android framework acts as a black box that hides the control flow of execution to static analyzers since they cannot afford to dive into the framework itself for scaling issues (cf. Section 1.2). Although our contributions show that there is a need to reveal, resolve, and model implicit calls, this is only a step toward holistic analyses. Indeed, the community needs to search the Android framework for delegation-like mechanisms systematically. Hence, the ambitious research agenda to systematically identify, with novel techniques, all possible delegation-like mechanisms in the Android framework in order to propose a more precise model of Android apps for static analyzers is inevitable.

Chapter 15

Conclusion

In this dissertation, we presented several approaches to address the challenge of analyzing code that is not accessible to existing Android apps' analyzers from two perspectives: dynamic and static analyzers.

The present dissertation is divided into two main parts: ① the identification and analysis of code that is difficult to access to dynamic analyzers, and ② the identification and analysis of code that is not accessible to existing static analyzers. These two objectives have been chosen with the aim of improving the overall comprehensiveness of static analysis for Android applications. A summary of each section is provided below.

The objective of the first part is to use static analysis to complement dynamic analysis. Indeed, dynamic analyzers miss many parts of the code because of insufficient input provided. In particular, Android framework mechanisms can be used to trigger logic bombs that are designed to evade dynamic analyzers with specific execution conditions, such as only being triggered when the device is connected to the Internet. In this manuscript, we contributed to the research endeavor by proposing the following: ① a replication study of a static logic bomb detector, for which we highlight the discrepancies between the original results and our observation with our prototype, TSOpen, that shows the limitations of this approach; ② a novel hybrid technique, Difuzer, combining code instrumentation, anomaly detection, and taint analysis to detect suspicious hidden sensitive operations toward triaging logic bombs in Android apps. Experimental results show that our proposed solution, Difuzer, can reveal up to 30% of logic bombs amongst suspicious hidden sensitive operations in real-world apps; and ③ in the same direction, we provided the research community with a new dataset of Android apps automatically infected with logic bombs for future research efforts to assess and compare new prototypes.

In the second part, we pursued an ambition to propose new techniques to improve the comprehensiveness of Android apps' static analysis. In particular, we first proposed a solution, RAICC, to account for atypical inter-component communication in Android apps, so far overlooked in the state of the art. We show that RAICC allows existing static data leak detectors to detect previously undetectable leaks and help find more ICC-related vulnerabilities. Then, we set up a novel approach, JuCify, toward unifying native code and bytecode in Android apps as a first step to account for the multi-language trend. Empirical results confirm that JuCify contributes to providing a better static model of Android apps and allows existing static data flow analyzers to propagate data flow values through native code. Ultimately, this manuscript describes our most recent work, Archer, to account for conditional implicit calls that hinder both static and dynamic analyzers. We thoroughly assess Archer on a qualitative view on a well-defined benchmark to show that it can reveal and resolve implicit calls to improve the comprehensiveness of Android apps' static analysis and resolve the condition needed to be met to trigger implicit calls

to aid dynamic analyzers cover particular parts of the code previously missed.

Overall, this dissertation contributes to the research field of Android apps' analysis by proposing novel approaches for statically analyzing code that was previously not accessible for existing static and dynamic analyses. Our approaches improve the comprehensiveness, the soundness, and the precision of static analysis and aid dynamic analysis in covering previously missed parts of the code.

Research Activities

In this chapter, we present the different research activities that were conducted throughout the Ph.D. journey. In particular, we list : ① the papers for which we contributed; ② the tools and datasets produced as an output to our research.

List of papers

In the following, we list the research papers for which we contributed during this Ph.D. thesis. Not all the papers were included in this manuscript.

Papers included in this dissertation:

- **[TDSC'21] Jordan Samhi**, Alexandre Bartel. On The (In)Effectiveness of Static Logic Bomb Detector for Android Apps. *IEEE Transactions on Dependable and Secure Computing*, 2021, 10.1109/TDSC.2021.3108057 [37].
- **[ICSE'21] Jordan Samhi**, Alexandre Bartel, Tegawendé F. Bissyandé, and Jacques Klein. RAICC: Revealing Atypical Inter-Component Communication in Android Apps. *In Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*, pages 1398-1409. IEEE, 2021, 10.1109/ICSE43902.2021.00126 [123].
- **[ICSE'22] Jordan Samhi**, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Difuzer: Uncovering Suspicious Hidden Sensitive Operations in Android Apps. *In Proceedings of the 44th IEEE/ACM International Conference on Software Engineering.* IEEE, 2022, 10.1145/3510003.3510135 [7].
- **[ICSE'22] Jordan Samhi**, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis. *In Proceedings of the 44th IEEE/ACM International Conference on Software Engineering.* IEEE, 2022, 10.1145/3510003.3512766 [27].
- **[MSR'22] Jordan Samhi**, Tegawendé F. Bissyandé, and Jacques Klein. TriggerZoo: A Dataset of Android Applications Automatically Infected with Logic Bombs. *In Proceedings of the 19th International Conference on Mining Software Repositories, Data Showcase.* 2022, 10.1145/3524842.3528020 [124].
- **[FSE'23] Jordan Samhi**, Ye Qiu, Rene Just, Michael D. Ernst, Tegawendé F. Bissyandé, and Jacques Klein. Archer: Resolving Conditional Implicit Calls for Comprehensive Analysis of Android Apps. *In peer review for the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* 2023.

Papers not included in this dissertation:

- [**MLHat'21**] Daoudi, Nadia, **Jordan Samhi**, Abdoul Kader Kabore, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. Dexray: A simple, yet effective deep learning approach to android malware detection based on image representation of bytecode. *In International Workshop on Deployable Machine Learning for Security Defense*, pages 81-106. Springer, 2021. 10.1007/978-3-030-87839-9_4 [28].
- [**EMSE'21**] **Jordan Samhi**, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. A first look at Android applications in Google Play related to COVID-19. *Empirical Software Engineering* pages 1-49, 2021. 10.1007/s10664-021-09943-x [247].
- [**TOSEM'22**] Xiaoyu Sun, Xiao Chen, Li Li, Haipeng Cai, John Grundy, **Jordan Samhi**, Tegawendé F. Bissyandé, Jacques Klein. Demystifying Hidden Sensitive Operations in Android Apps. ACM Transactions on Software Engineering and Methodology [248].
- [**SANER'23**] **Jordan Samhi**, Maria Kober, Abdoul Kader Kabore, Steven, Arzt, Tegawendé F. Bissyandé, Jacques Klein. Negative Results of Fusing Code and Documentation for Learning to Accurately Identify Sources and Sinks in the Android Framework for Sensitive Data Flow Analysis. *In Proceedings of the 30th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering, RENE track, 2023.*
- [**MobileSoft'23**] Maria Kober, **Jordan Samhi**, Steven Arzt, Tegawendé F. Bissyandé, and Jacques Klein. Sensitive and Personal Data: What Exactly Are You Talking About?. *In peer review for the 10th International Conference on Mobile Software Engineering and Systems 2023, NIER track.*
- [**FSE'23**] Xiaoyu Sun, **Jordan Samhi**, Xiaobin hu, Li Li, John Grundy. DroidTEC: Understanding and Detecting Typestate Misuse in Android Applications. *In peer review for the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2023.

Tools and datasets

In the following, we list the tools and datasets that were produced throughout this Ph.D. thesis.

Tools:

- RAICC: `https://github.com/JordanSamhi/RAICC`
- TSOpen: `https://github.com/JordanSamhi/TSOpen`
- Difuzer: `https://github.com/JordanSamhi/Difuzer`
- JuCify: `https://github.com/JordanSamhi/JuCify`
- Javalyzer: `https://github.com/JordanSamhi/Javalyzer`
- AndroBomb: `https://github.com/JordanSamhi/AndroBomb`
- Archer: `https://github.com/JordanSamhi/Archer`
- CoDoC: `https://github.com/JordanSamhi/CoDoC`

Existing tools updated and improved:

- IC3: `https://github.com/JordanSamhi/ic3`
- COAL: `https://github.com/JordanSamhi/coal`
- COAL-Strings: `https://github.com/JordanSamhi/coal-strings`

Datasets:

- APKCovid: `https://github.com/JordanSamhi/APKCOVID`
- TriggerZoo: `https://github.com/JordanSamhi/TriggerZoo`
- SensitiveData: `https://github.com/JordanSamhi/SensitiveData`
- Benchmarks:
  - RAICC: `https://github.com/JordanSamhi/RAICC/tree/master/artefacts/droidbench`
  - JuCify: `https://github.com/JordanSamhi/JuCify/tree/master/benchApps`
  - Archer: `https://github.com/JordanSamhi/Archer/tree/main/artefacts/rq3/benchmark_apps`

Bibliography

[1] J. Greig, "Mcafee/fireeye merger completed, ceo says automation only way forward for cybersecurity," 2021, accessed October 2022. [Online]. Available: https://www.zdnet.com/article/mcafeefireeye-merger-completed-ceo-says-automation-only-way-forward-for-cybersecurity/

[2] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "Anastasia: Android malware detection using static analysis of applications," in *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2016, pp. 1–5.

[3] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1092–1104. [Online]. Available: https://doi.org/10.1145/2660267.2660343

[4] H. Kang, J. wook Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 479174, 2015. [Online]. Available: https://doi.org/10.1155/2015/479174

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN NOTICES*, vol. 49, no. 6, p. 259–269, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[6] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 280–291.

[7] J. Samhi, L. Li, T. F. Bissyande, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 723–735. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3510135

[8] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New

York, NY, USA: Association for Computing Machinery, 2016, p. 318–329. [Online]. Available: https://doi.org/10.1145/2931037.2931044

[9] H. Wu, S. Yang, and A. Rountev, "Static detection of energy defect patterns in android applications," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 185–195. [Online]. Available: https://doi.org/10.1145/2892208.2892218

[10] L. Pan, B. Cui, H. Liu, J. Yan, S. Wang, J. Yan, and J. Zhang, *Static Asynchronous Component Misuse Detection for Android Applications*. New York, NY, USA: Association for Computing Machinery, 2020, p. 952–963. [Online]. Available: https://doi.org/10.1145/3368089.3409699

[11] F. Wei, S. Roy, X. 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*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1329–1341. [Online]. Available: https://doi.org/10.1145/2660267.2660357

[12] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of android malware," in *Proceedings of the Seventh European Workshop on System Security*, ser. EuroSec '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592791.2592796

[13] V. Van Der Veen, H. Bos, and C. Rossow, "Dynamic analysis of android malware," *Internet & Web Technology Master thesis, VU University Amsterdam*, 2013.

[14] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 93–104. [Online]. Available: https://doi.org/10.1145/2381934.2381950

[15] M. Zheng, M. Sun, and J. C. S. Lui, "Droidtrace: A ptrace based android dynamic analysis system with forward execution capability," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2014, pp. 128–133.

[16] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2619091

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

[18] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving Java reflection and Android intents," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, November 11–13 2015, pp. 669–679.

[19] CyberTalk, "10 eye-opening mobile malware statistics to know," 2022, accessed October 2022. [Online]. Available: https://www.cybertalk.org/2022/06/10/10-eye-opening-mobile-malware-statistics-to-know/

[20] D. Palmer, "This powerful android malware stayed hidden for years, infecting tens of thousands of smartphones," 2020, accessed October 2022. [Online]. Available: https://www.zdnet.com/article/this-powerful-android-malware-stayed-hidden-years-infected-tens-of-thousands-of-smartphones/

[21] S. T. Intelligence and R. Team, "Poseidon's offspring: Charybdis and scylla," 2022, accessed October 2022. [Online]. Available: https://www.humansecurity.com/learn/blog/poseidons-offspring-charybdis-and-scylla

[22] B. Toulas, "New android malware apps installed 10 million times from google play," 2022, accessed October 2022. [Online]. Available: https://www.bleepingcomputer.com/news/security/new-android-malware-apps-installed-10-million-times-from-google-play/

[23] G. Cluley, "More malware-infested apps found in the google play store," 2022, accessed October 2022. [Online]. Available: https://www.tripwire.com/state-of-security/security-data-protection/cyber-security/more-malware-infested-apps-found-google-play-store/

[24] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[25] F. E. Allen, "Control flow analysis," *ACM SIGPLAN NOTICES*, vol. 5, no. 7, p. 1–19, Jul. 1970. [Online]. Available: https://doi.org/10.1145/390013.808479

[26] S. Arzt, S. Rasthofer, and E. Bodden, "Instrumenting android and java applications as easy as abc," in *Runtime Verification*, A. Legay and S. Bensalem, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 364–381.

[27] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1232–1244. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3512766

[28] N. Daoudi, J. Samhi, A. K. Kabore, K. Allix, T. F. Bissyandé, and J. Klein, "Dexray: A simple, yet effective deep learning approach to android malware detection based on image representation of bytecode," in *Deployable Machine Learning for Security Defense*, G. Wang, A. Ciptadi, and A. Ahmadzadeh, Eds. Cham: Springer International Publishing, 2021, pp. 81–106.

[29] L. Xu, D. Zhang, N. Jayasena, and J. Cavazos, "Hadm: Hybrid analysis for detection of malware," in *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*, Y. Bi, S. Kapoor, and R. Bhatia, Eds. Cham: Springer International Publishing, 2018, pp. 702–724.

[30] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, *Automatically Identifying Trigger-based Behavior in Malware*. Boston, MA: Springer US, 2008, pp. 65–88. [Online]. Available: https://doi.org/10.1007/978-0-387-68768-1%5F4

[31] M. Choudhary and B. Kishore, "Haamd: Hybrid analysis for android malware detection," in *2018 International Conference on Computer Communication and Informatics (ICCCI)*, 2018, pp. 1–4.

[32] H. Berger, C. Hajaj, and A. Dvir, "Evasion is not enough: A case study of android malware," in *Cyber Security Cryptography and Machine Learning*, S. Dolev, V. Kolesnikov, S. Lodha, and G. Weiss, Eds. Cham: Springer International Publishing, 2020, pp. 167–174.

[33] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Security and Privacy in Communication Networks*, R. Beyah, B. Chang, Y. Li, and S. Zhu, Eds. Cham: Springer International Publishing, 2018, pp. 172–192.

[34] P. Chen, L. Desmet, and C. Huygens, "A study on advanced persistent threats," in *Communications and Multimedia Security*, B. De Decker and A. Zúquete, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 63–72.

[35] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, Jul. 2009. [Online]. Available: https://doi.org/10.1145/1541880.1541882

[36] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Computation*, vol. 13, no. 7, pp. 1443–1471, 2001. [Online]. Available: https://doi.org/10.1162/089976601750264965

[37] J. Samhi and A. Bartel, "On the (in)effectiveness of static logic bomb detector for android apps," *IEEE Transactions on Dependable and Secure Computing*, no. 01, pp. 1–1, Aug. 2021.

[38] StatCounter, "Mobile operating system market share worldwide, https://gs.statcounter.com/os-market-share/mobile/worldwide," 2022, accessed June 2022.

[39] Google, "Google play protect," 2022. [Online]. Available: https://www.android.com/play-protect/

[40] J.-T. Chan and W. Yang, "Advanced obfuscation techniques for java bytecode," *Journal of Systems and Software*, vol. 71, no. 1, pp. 1–10, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121202000663

[41] S. Liang and G. Bracha, "Dynamic class loading in the java virtual machine," *ACM SIGPLAN NOTICES*, vol. 33, no. 10, p. 36–44, Oct. 1998. [Online]. Available: https://doi.org/10.1145/286942.286945

[42] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for java web applications," in *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*. Berlin, Heidelberg: Springer-Verlag, 2014, p. 140–154. [Online]. Available: https://doi.org/10.1007/978-3-642-54804-8%5F10

[43] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed android applications," in *Computer Security – ESORICS 2015*, G. Pernul, P. Y A Ryan, and E. Weippl, Eds. Cham: Springer International Publishing, 2015, pp. 293–311.

[44] D. Papp, L. Buttyán, and Z. Ma, "Towards semi-automated detection of trigger-based behavior for software security assurance," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3098954.3105821

[45] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," *no*, 1998.

[46] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[47] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 625–642.

[48] G. Phipps, "Comparing observed bug and productivity rates for java and c++," *Software: Practice and Experience*, vol. 29, no. 4, p. 345–358, Apr. 1999.

[49] A. Bartel, J. Klein, Y. Le Traon, and M. 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*, ser. SOAP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 27–38. [Online]. Available: https://doi.org/10.1145/2259051.2259056

[50] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, ser. CC'03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 153–169.

[51] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 217–228. [Online]. Available: https://doi.org/10.1145/2382196.2382222

[52] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," *University of Darmstadt*, 2013.

[53] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic hpc cluster: The ul experience," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 959–967.

[54] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: Appropriate use and interpretation," *Anesthesia & Analgesia*, vol. 126, no. 5, 2018. [Online]. Available: https://journals.lww.com/anesthesia-analgesia/Fulltext/2018/05000/Correlation%5FCoefficients%5F%5FAppropriate%5FUse%5Fand.50.aspx

[55] V. Total, "Virustotal-free online virus, malware and url scanner," 2022.

[56] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in android apps," in *2016 IEEE 23rd International Conference*

on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, 2016, pp. 403–414.

[57] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, M. Tokoro and R. Pareschi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 77–101.

[58] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 324–341. [Online]. Available: https://doi.org/10.1145/236337.236371

[59] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, *Practical Virtual Method Call Resolution for Java.* New York, NY, USA: Association for Computing Machinery, 2000, p. 264–280. [Online]. Available: https://doi.org/10.1145/353171.353189

[60] X. Jiang and Y. Zhou, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy.* Los Alamitos, CA, USA: IEEE Computer Society, May 2012, pp. 95–109. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2012.16

[61] T. Seals, "Cerberus enters the android malware rental scene," 2019.

[62] M. Stone, "The path to the payload: Android edition," 2019.

[63] K. Sun, "Google play apps drop anubis banking malware, use motion-based evasion tactics," 2019.

[64] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps," in *NDSS*, 2017.

[65] C. Cadar, D. Dunbar, and D. 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*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[66] L. Bello and M. Pistoia, "Ares: Triggering payload of evasive android malware," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2–12. [Online]. Available: https://doi.org/10.1145/3197231.3197239

[67] Q. Zeng, L. Luo, Z. Qian, X. Du, Z. Li, C.-T. Huang, and C. Farkas, "Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, p. 2582–2600, Nov. 2021. [Online]. Available: https://doi.org/10.1109/TDSC.2019.2957787

[68] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing android apps," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 550–561.

[69] C. Cimpanu, "Play store identified as main distribution vector for most android malware," 2021, accessed February 2021. [Online]. Available: https://www.zdnet.com/article/play-store-identified-as-main-distribution-vector-for-most-android-malware/

[70] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1106–1120.

[71] V. Van Der Veen, H. Bos, and C. Rossow, "Dynamic analysis of android malware," *Internet & Web Technology Master thesis, VU University Amsterdam*, 2013.

[72] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *2012 European Intelligence and Security Informatics Conference*, 2012, pp. 141–147.

[73] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, 2013, pp. 300–305.

[74] E. Erturk, "A case study in open source software security and privacy: Android adware," in *World Congress on Internet Security (WorldCIS-2012)*, 2012, pp. 189–191.

[75] H. Pieterse and M. S. Olivier, "Android botnets on the rise: Trends and characteristics," in *2012 Information Security for South Africa*, 2012, pp. 1–5.

[76] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated detection and analysis for android ransomware," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, 2015, pp. 1338–1343.

[77] M. H. Saad, A. Serageldin, and G. I. Salama, "Android spyware disease and medication," in *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*, 2015, pp. 118–125.

[78] W. Zhou, X. Zhang, and X. Jiang, "Appink: Watermarking android apps for repackaging deterrence," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1–12. [Online]. Available: https://doi.org/10.1145/2484313.2484315

[79] L. Li, T. F. Bissyande, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Transactions on Software Engineering*, vol. 47, no. 04, pp. 676–693, Apr. 2021.

[80] L. Li, T. F. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 136–143.

[81] O. Gadyatskaya, A.-L. Lezza, and Y. Zhauniarovich, "Evaluation of resource-based app repackaging detection in android," in *Secure IT Systems*, B. B. Brumley and J. Röning, Eds. Cham: Springer International Publishing, 2016, pp. 135–151.

[82] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953. [Online]. Available: http://www.jstor.org/stable/1990888

[83] H. Agrawal, J. Alberi, L. Bahler, J. Micallef, A. Virodov, M. Magenheimer, S. Snyder, V. Debroy, and E. Wong, "Detecting hidden logic bombs in critical infrastructure software," in *International Conference on Information Warfare and Security. Academic Conferences International Limited*, vol. 1, 2012.

[84] O. Topgul, "Android malware evasion techniques - emulator detection," 2022, accessed June 2022. [Online]. Available: https://www.oguzhantopgul.com/2014/12/android-malware-evasion-techniques.html

[85] S. Alexander-Bown, "Android security: Adding tampering detection to your app," 2022, accessed February 2021. [Online]. Available: https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app#4-1-emulator

[86] H. Dharmdasani, "Android.hehe: Malware now disconnects phone calls," 2022, accessed June 2022. [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html

[87] T. Micro, "Hacking team spying tool listens to calls," 2021, accessed February 2021. [Online]. Available: https://www.trendmicro.com/en%5Fus/research/15/g/hacking-team-rcsandroid-spying-tool-listens-to-calls-roots-devices-to-get-in.html

[88] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of android taint-analysis results," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 102–114. [Online]. Available: https://doi.org/10.1109/ASE.2019.00020

[89] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, "Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps." in *NDSS*, 2018.

[90] M. Junaid, D. Liu, and D. Kung, "Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models," *computers & security*, vol. 59, no. C, p. 92–117, Jun. 2016. [Online]. Available: https://doi.org/10.1016/j.cose.2016.01.008

[91] Google, "Context class, https://developer.android.com/reference/android/ content/context#getsystemservice(java.lang.string)," 2022, accessed June 2022.

[92] ——, "Sensormanager class, https://developer.android.com/reference/android/hardware/sensormanager," 2022, accessed June 2022.

[93] ——, "Sensor class, https://developer.android.com/reference/android/ hardware/sensor," 2022, accessed June 2022.

[94] ——, "Cursor class, https://developer.android.com/reference/android/database/cursor," 2022, accessed June 2022.

[95] ——, "Build class, https://developer.android.com/reference/android/os/Build," 2022, accessed June 2022.

[96] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Trust*

and Trustworthy Computing, S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 291–307.

[97] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in EHealth, HCI, Information Retrieval and Pervasive Technologies*. NLD: IOS Press, 2007, p. 3–24.

[98] V. N. Vapnik, "An overview of statistical learning theory," *IEEE Transactions on Neural Networks*, vol. 10, no. 5, pp. 988–999, 1999.

[99] H. Xu, C. Caramanis, and S. Mannor, "Robustness and regularization of support vector machines." *Journal of machine learning research*, vol. 10, no. 7, 2009.

[100] Yunqiang Chen, Xiang Sean Zhou, and T. S. Huang, "One-class svm for learning in image retrieval," in *Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205)*, vol. 1, 2001, pp. 34–37 vol.1.

[101] Li, D. Li, T. F. Bissyande, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, p. 1269–1284, Jun. 2017. [Online]. Available: https://doi.org/10.1109/TIFS.2017.2656460

[102] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative: Automating and optimizing detection of android native code malware variants," *Computers & Security*, vol. 65, pp. 230–246, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016740481630164X

[103] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden, "How current android malware seeks to evade automated code analysis," in *Information Security Theory and Practice*, R. N. Akram and S. Jajodia, Eds. Cham: Springer International Publishing, 2015, pp. 187–202.

[104] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917302987

[105] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, May 2011. [Online]. Available: https://doi.org/10.1145/1961189.1961199

[106] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou, "Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios," in *2016 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2016, pp. 357–376. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2016.29

[107] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, p. 1529–1544, Jul. 2017. [Online]. Available: https://doi.org/10.1109/TIFS.2017.2661723

[108] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," *ACM SIGPLAN NOTICES*, vol. 49, no. 6, p. 316–325, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594311

[109] C.-M. Lin, J.-H. Lin, C.-R. Dow, and C.-M. Wen, "Benchmark dalvik and native code for android system," in *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, 2011, pp. 320–323.

[110] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *2008 IEEE International Conference on Dependable Systems & Networks With FTCS and DCC (DSN)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2008. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/DSN.2008.4630086

[111] D. Shi, X. Tang, and Z. Ye, "Detecting environment-sensitive malware based on taint analysis," in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2017, pp. 322–327.

[112] X. Jia, G. Zhou, Q. Huang, W. Zhang, and D. Tian, "Findevasion: An effective environment-sensitive malware detection system for the cloud," in *Digital Forensics and Cyber Crime*, P. Matoušek and M. Schmiedecker, Eds. Cham: Springer International Publishing, 2018, pp. 3–17.

[113] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Recent Advances in Intrusion Detection*, R. Sommer, D. Balzarotti, and G. Maier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 338–357.

[114] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 287–301.

[115] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient detection of split personalities in malware," in *NDSS 2010, 17th Annual Network and Distributed System Security Symposium, February 28th-March 3rd, 2010, San Diego, USA*, ISOC, Ed., San Diego, 2010, iSOC. Personal use of this material is permitted. The definitive version of this paper was published in NDSS 2010, 17th Annual Network and Distributed System Security Symposium, February 28th-March 3rd, 2010, San Diego, USA and is available at :.

[116] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 426–436.

[117] I. Burguera, U. Zurutuza, and S. 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*, ser. SPSM '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 15–26. [Online]. Available: https://doi.org/10.1145/2046614.2046619

[118] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer, "Andrubis – 1,000,000 apps later: A view on current android malware behaviors," in *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014, pp. 3–17.

[119] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *Ndss*, 2015.

[120] A. Mahindru and P. Singh, "Dynamic permissions based android malware detection using machine learning techniques," in *Proceedings of the 10th Innovations in Software Engineering Conference*, ser. ISEC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 202–210. [Online]. Available: https://doi.org/10.1145/3021460.3021485

[121] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn, "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 301–308. [Online]. Available: https://doi.org/10.1145/3029806.3029823

[122] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 576–587. [Online]. Available: https://doi.org/10.1145/2635868.2635869

[123] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 1398–1409. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00126

[124] J. Samhi, T. F. Bissyande, and J. Klein, "Triggerzoo: A dataset of android applications automatically infected with logic bombs," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 459–463. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3524842.3528020

[125] S. Nielebock, P. Blockhaus, J. Krüger, and F. Ortmeier, "Androidcompass: A dataset of android compatibility checks in code repositories," in *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) - Data Showcase Track*, IEEE, Ed., 2021. [Online]. Available: https://arxiv.org/abs/2103.09620https://doi.org/10.5281/zenodo.4428340https://www.youtube.com/watch?v=M3ruWediurs

[126] T. Wendland, J. Sun, J. Mahmud, S. M. H. Mansur, S. Huang, K. Moran, J. Rubin, and M. Fazzini, "Andror2: A dataset of manually-reproduced bug reports for android apps," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 600–604.

[127] W. Li, X. Fu, and H. Cai, "Androct: Ten years of app call traces in android," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 570–574.

[128] Google, "Android ndk," 2022. [Online]. Available: https://developer.android.com/ndk

[129] ——, "Zipalign," 2022. [Online]. Available: https://developer.android.com/studio/command-line/zipalign

[130] ——, "Sign your app," 2022. [Online]. Available: https://developer.android.com/studio/publish/app-signing

[131] M. Fan, L. Yu, S. Chen, H. Zhou, X. Luo, S. Li, Y. Liu, J. Liu, and T. Liu, "An empirical evaluation of gdpr compliance violations in android mhealth apps," in *ISSRE*, 2020, pp. 253–264.

[132] X. Sun, L. Li, T. F. Bissyandé, J. Klein, D. Octeau, and J. Grundy, "Taming reflection: An essential step toward whole-program analysis of android apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, apr 2021. [Online]. Available: https://doi.org/10.1145/3440033

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

[134] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 239–252. [Online]. Available: https://doi.org/10.1145/1999995.2000018

[135] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, 2015, p. 110.

[136] F. I. Abro, M. Rajarajan, T. M. Chen, and Y. Rahulamathavan, "Android application collusion demystified," in *Future Network Systems and Security*, R. Doss, S. Piramuthu, and W. Zhou, Eds. Cham: Springer International Publishing, 2017, pp. 176–187.

[137] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, p. 1252–1264, Jun. 2016. [Online]. Available: https://doi.org/10.1109/TIFS.2016.2523912

[138] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. USA: USENIX Association, 2013, p. 543–558.

[139] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 77–88.

[140] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving java reflection and android intents," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 669–679. [Online]. Available: https://doi.org/10.1109/ASE.2015.69

[141] Google, "Services documentation," 2020. [Online]. Available: https://developer.android.com/guide/components/services

[142] ——, "Components fundamentals documentation," 2020. [Online]. Available: https://developer.android.com/guide/components/fundamentals

[143] ——, "Intents documentation," 2020. [Online]. Available: https://developer.android.com/training/basics/intents

[144] S. S. Engineering, "Droidbench: Open-source test suite," 2022. [Online]. Available: https://github.com/secure-software-engineering

[145] S. Groß, A. Tiwari, and C. Hammer, "Pianalyzer: A precise approach for pending-intent vulnerability analysis," in *Computer Security*, J. Lopez, J. Zhou, and M. Soriano, Eds. Cham: Springer International Publishing, 2018, pp. 41–59.

[146] Google, "Pendingintent documentation," 2022. [Online]. Available: https://developer.android.com/reference/android/app/PendingIntent

[147] ——, "Intentsender documentation," 2022. [Online]. Available: https://developer.android.com/reference/android/content/IntentSender

[148] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 241–250. [Online]. Available: https://doi.org/10.1145/1985793.1985827

[149] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Apkcombiner: Combining multiple android apps to support inter-app analysis," in *ICT Systems Security and Privacy Protection*, H. Federrath and D. Gollmann, Eds. Cham: Springer International Publishing, 2015, pp. 513–527.

[150] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 421–431. [Online]. Available: https://doi.org/10.1145/3180155.3180228

[151] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 242–251. [Online]. Available: https://doi.org/10.1145/2597073.2597109

[152] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "Scandal: Static analyzer for detecting privacy leaks in android applications," *MoST*, vol. 12, no. 110, p. 1, 2012.

[153] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1457–1462. [Online]. Available: https://doi.org/10.1145/2245276.2232009

[154] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *2012 Third World Congress on Software Engineering*, 2012, pp. 101–104.

[155] L. Li, K. Allix, D. Li, A. Bartel, T. F. Bissyandé, and J. Klein, "Potential component leaks in android apps: An investigation into a new feature set for malware detection," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 195–200.

[156] A. Jha, S. Lee, and W. Lee, "Modeling and test case generation of inter-component communication in android," in *2015 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2015, pp. 113–116. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MobileSoft.2015.24

[157] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security and Privacy*, vol. 7, no. 1, p. 50–57, Jan. 2009. [Online]. Available: https://doi.org/10.1109/MSP.2009.26

[158] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard – enforcing user requirements on android apps," in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Piterman and S. A. Smolka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 543–548.

[159] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 639–652. [Online]. Available: https://doi.org/10.1145/2046707.2046780

[160] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 539–552. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu%5Frubin

[161] A. Bartel, J. Klein, M. Monperrus, K. Allix, and y. Le Traon, "Improving privacy on android smartphones through in-vivo bytecode instrumentation," University of Luxembourg, Research Report arXiv:1208.4536, 2012. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00700319

[162] G. S. Babil, O. Mehani, R. Boreli, and M.-A. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices," in *2013 International Conference on Security and Cryptography (SECRYPT)*, 2013, pp. 1–8.

[163] M. Hammad, J. Garcia, and S. Malek, "Self-protection of android systems from inter-component communication attacks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 726–737. [Online]. Available: https://doi.org/10.1145/3238147.3238207

[164] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek, "Automatic generation of inter-component communication exploits for android applications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 661–671. [Online]. Available: https://doi.org/10.1145/3106237.3106286

[165] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *The Network and Distributed System Security Symposium*, 2016, pp. 1–15.

[166] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1137–1150. [Online]. Available: https://doi.org/10.1145/3243734.3243835

[167] C. Qian, X. Luo, Y. Shao, and A. S. Chan, "On tracking information flows through jni in android applications," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2014, pp. 180–191. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/DSN.2014.30

[168] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, Aug. 2006. [Online]. Available: https://doi.org/10.1007/s11416-006-0012-2

[169] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, ser. WiSec '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 165–176. [Online]. Available: https://doi.org/10.1145/2627393.2627396

[170] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 331–342. [Online]. Available: https://doi.org/10.1145/2976749.2978343

[171] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 127–137. [Online]. Available: https://doi.org/10.1145/3324884.3416558

[172] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis, "Identifying java calls in native code via binary scanning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 388–400. [Online]. Available: https://doi.org/10.1145/3395363.3397368

[173] JNI, "https://docs.oracle.com/javase/8/docs/technotes/guides/jni/," 2022, accessed June 2022.

[174] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 62–72. [Online]. Available: https://doi.org/10.1145/1065010.1065019

[175] J. Functions, "https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html," 2022, accessed June 2022.

[176] I. R. Forman, N. Forman, and J. V. Ibm, *Java reflection in action*. Citeseer, 2004.

[177] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, p. 63–68, Dec. 2005. [Online]. Available: https://doi.org/10.1145/1127577.1127590

[178] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, pp. 233–245, Nov. 2011. [Online]. Available: https://doi.org/10.1007/s11416-011-0151-y

[179] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 2, p. 158–191, Apr. 1998. [Online]. Available: https://doi.org/10.1145/279310.279314

[180] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2016, pp. 138–157. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2016.17

[181] Androguard, "https://androguard.readthedocs.io," 2022, accessed June 2022.

[182] N. S. G. page, "https://github.com/plast-lab/native-scanner," 2022, accessed June 2022.

[183] D. G. page, "https://bitbucket.org/yanniss/doop/src/master/," 2022, accessed June 2022.

[184] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 03 1947. [Online]. Available: https://doi.org/10.1214/aoms/1177730491

[185] XAMARIN, "https://dotnet.microsoft.com/apps/xamarin," 2022, accessed June 2022.

[186] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 316–325. [Online]. Available: https://doi.org/10.1145/2594291.2594311

[187] Skylot, "Jadx: Dex to java decompiler, https://github.com/skylot/jadx," 2022, accessed June 2022.

[188] J. Křoustek and P. Matula, "Retdec: An open-source machine-code decompiler," [talk], July 2018, presented at Pass the SALT 2018, Lille, FR.

[189] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 281–294. [Online]. Available: https://doi.org/10.1145/2307636.2307663

[190] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *2012 Seventh Asia Joint Conference on Information Security (ASIA JCIS)*. Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2012, pp. 62–69. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/AsiaJCIS.2012.18

[191] L. Cruz, R. Abreu, J. Grundy, L. Li, and X. Xia, "Do energy-oriented changes hinder maintainability?" in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 29–40.

[192] P. Kong, L. Li, J. Gao, T. F. Bissyandé, and J. Klein, *Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems*. New York, NY, USA: Association for Computing Machinery, 2019, p. 78–89. [Online]. Available: https://doi.org/10.1145/3293882.3330572

[193] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 187–198. [Online]. Available: https://doi.org/10.1145/3180155.3180243

[194] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2019.

[195] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 94–105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[196] H. Zhang, H. Wu, and A. Rountev, "Automated test generation for detection of leaks in android applications," in *Proceedings of the 11th International Workshop on Automation of Software Test*, ser. AST '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 64–70. [Online]. Available: https://doi.org/10.1145/2896921.2896932

[197] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 245–256. [Online]. Available: https://doi.org/10.1145/3106237.3106298

[198] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari, "Static analysis of binary code to isolate malicious behaviors," in *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, ser. WETICE '99. USA: IEEE Computer Society, 1999, p. 184–189.

[199] A. Fraboulet and C. Cifuentes, "Intraprocedural static slicing of binary executables," in *2013 IEEE International Conference on Software Maintenance*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1997, p. 188. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSM.1997.624245

[200] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, Aug. 2014. [Online]. Available: https://doi.org/10.1007/s11416-014-0203-1

[201] L. Li and C. Wang, "Dynamic analysis and debugging of binary code for security applications," in *Runtime Verification*, A. Legay and S. Bensalem, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 403–423.

[202] Y.-H. Choi, M.-W. Park, J.-H. Eom, and T.-M. Chung, "Dynamic binary analyzer for scanning vulnerabilities with taint analysis," *Multimedia Tools and Applications*, vol. 74, no. 7, pp. 2301–2320, Apr. 2015. [Online]. Available: https://doi.org/10.1007/s11042-014-1922-5

[203] K. A. Roundy and B. P. Miller, "Hybrid analysis and control of malware," in *Recent Advances in Intrusion Detection*, S. Jha, R. Sommer, and C. Kreibich, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 317–338.

[204] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, Feb. 2017. [Online]. Available: https://doi.org/10.1007/s11416-015-0261-z

[205] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu, "Binmatch: A semantics-based hybrid approach on binary code clone analysis," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 104–114.

[206] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *KSII The 9th International Conference on Internet (ICONI) 2017 Symposium*, 2017.

[207] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 388–398.

[208] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "Typeminer: Recovering types in binary programs using machine learning," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, pp. 288–308.

[209] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98.

[210] Radare2, "https://github.com/radareorg/radare2," 2022, accessed June 2022.

[211] Z. Kan, H. Wang, L. Wu, Y. Guo, and D. X. Luo, "Automated deobfuscation of android native binary code," *arXiv preprint*, 2019.

[212] C. Rizzo, "Static flow analysis for hybrid and native android applications," Ph.D. dissertation, Royal Holloway – University of London, 2020.

[213] IDC, "Smartphone market share, https://www.idc.com/promo/smartphone-market-share/os," accessed September 2022.

[214] M. Iqbal, "App download data, https://www.businessofapps.com/data/app-statistics/," accessed April 2022.

[215] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid source code, https://github.com/secure-software-engineering/FlowDroid/blob/develop/soot-infoflow/src/soot/jimple/infoflow/cfg/LibraryClassPatcher.java," 2022, accessed April 2022.

[216] B. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 216–226, 1979.

[217] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," *ACM SIGPLAN NOTICES*, vol. 31, no. 10, p. 324–341, Oct. 1996. [Online]. Available: https://doi.org/10.1145/236338.236371

[218] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, Cornell, 1994.

[219] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 32–41. [Online]. Available: https://doi.org/10.1145/237721.237727

[220] M. Cao, K. Ahmed, and J. Rubin, "Rotten apples spoil the bunch: An anatomy of google play malware," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1919–1931. [Online]. Available: https://doi.org/10.1145/3510003.3510161

[221] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, ser. SOAP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 3–8. [Online]. Available: https://doi.org/10.1145/2259051.2259052

[222] S. Researcher, "Stack overflow, https://stackoverflow.com/q/70670020/," Jan. 2022, accessed August 2022.

[223] Google, "Schedule tasks with workmanager," 2022, accessed August 2022. [Online]. Available: https://developer.android.com/topic/libraries/architecture/workmanager

[224] ——, "Android guides, https://developer.android.com/guide," accessed August 2022.

[225] ——, "Android guide," 2021, accessed April 2022. [Online]. Available: https://developer.android.com/guide/background/threading

[226] ——, "Android guide," 2022, accessed April 2022. [Online]. Available: https://developer.android.com/guide/background

[227] ——, "Android guide," 2021, accessed April 2022. [Online]. Available: https://developer.android.com/topic/libraries/architecture/workmanager/migrating-fb

[228] ——, "Android guide," 2022, accessed April 2022. [Online]. Available: https://developer.android.com/topic/libraries/architecture/workmanager

[229] ——, "Android guide," 2021, accessed April 2022. [Online]. Available: https://developer.android.com/topic/performance/background-optimization

[230] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 49–61. [Online]. Available: https://doi.org/10.1145/199448.199462

[231] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, jul 2020. [Online]. Available: https://doi.org/10.1145/3395042

[232] Google, "Ui/application exerciser monkey," 2022, accessed August 2022. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[233] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.

[234] J. Yan, S. Zhang, Y. Liu, J. Yan, and J. Zhang, "Iccbot: Fragment-aware and context-sensitive icc resolution for android applications," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 105–109.

[235] S. Chen, L. Fan, C. Chen, and Y. Liu, "Automatically distilling storyboard with rich features for android apps," *IEEE Transactions on Software Engineering*, 2022.

[236] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 596–607.

[237] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for android apps," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1054–1076, 2016.

[238] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 89–99.

[239] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 658–668. [Online]. Available: https://doi.org/10.1109/ASE.2015.76

[240] W. You, B. Liang, J. Li, W. Shi, and X. Zhang, "Android implicit information flow demystified," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 585–590. [Online]. Available: https://doi.org/10.1145/2714576.2714604

[241] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 252–276.

[242] SentinelOne, "Hiding code inside images: How malware uses steganography," 2019, accessed October 2022. [Online]. Available: https://www.sentinelone.com/blog/hiding-code-inside-images-malware-steganography/

[243] Google, "Flutter," 2022, accessed October 2022. [Online]. Available: https://flutter.dev

[244] D. Co., "Ionic framework," 2022, accessed October 2022. [Online]. Available: https://ionicframework.com

[245] Meta, "React native," 2022, accessed October 2022. [Online]. Available: https://reactnative.dev

[246] A. S. Foundation, "Cordova," 2022, accessed October 2022. [Online]. Available: https://cordova.apache.org

[247] J. Samhi, K. Allix, T. F. Bissyandé, and J. Klein, "A first look at android applications in google play related to covid-19," *Empirical Software Engineering*, vol. 26, no. 4, p. 57, Apr. 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-09943-x

[248] X. Sun, X. Chen, L. Li, H. Cai, J. Grundy, J. Samhi, T. F. Bissyandé, and J. Klein, "Demystifying hidden sensitive operations in android apps," *ACM Transactions on Software Engineering and Methodology*, 2022.