NAGOYA INSTITUTE OF TECHNOLOGY

DOCTORAL THESIS

A Study on Taint Analysis with Runtime Data for Tracking Information Flows in Android Apps

(Android アプリ内情報フローを追跡する実行時データを用いたテイント解析に関する研究)

Author:
Hiroki INAYOSHI

Supervisor:
Prof. Shoichi SAITO

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Engineering

in the

Department of Computer Science

January 16, 2024

NAGOYA INSTITUTE OF TECHNOLOGY

Abstract

Department of Computer Science

Doctor of Engineering

A Study on Taint Analysis with Runtime Data for Tracking Information Flows in Android Apps

by Hiroki INAYOSHI

Toward a reliable analysis of real-world Android apps, this thesis proposes two novel taint analysis tools named VTDroid and T-Recs based on the idea of utilizing the app's runtime data.

Chapter 1 This chapter introduces Android OS, which occupies 70% of the total mobile OS market share in 2023. Then, it discusses the increasing need to protect user privacy. It also explains a need to uncover how well app developers and third-party SDK providers follow the privacy protection rules. Researchers have investigated real-world apps and found many non-compliant, policy-violating, and protection-circumventing behaviors. Taint analysis techniques have been actively developed and utilized to detect such suspicious behaviors. Finally, this chapter briefly summarizes two taint-analysis-related issues addressed by this thesis and the approaches.

Chapter 2 Chapter 2 describes the fundamentals for understanding taint analysis for Android apps. First, it explains information flow types, such as direct assignment and control dependence. Then, it discusses Android- and Java-specific features that are keys in the Android app analysis. Finally, it introduces taint analysis of Android apps.

Chapter 3 Taint analysis can be circumvented by anti-taint-analysis (ATA) techniques. A series of ATA techniques has been demonstrated on the Android platform. They are only a few lines of code each and could be introduced into apps with obfuscator tools by app developers to defend their apps against a taint analysis. However, there are only a few counter approaches against ATA techniques, which are only partially effective against ATA techniques.

Chapter 3 proposes VTDroid, which is designed to make it difficult for apps to evade taint tracking by neutralizing uncomplicated techniques not specific to a particular ATA technique. This chapter characterizes the ATA techniques by four types of information flow. It proposes value logging and matching that propagate taint among registers based on their data values, in addition to the traditional bytecode-level tracking. VTDroid is evaluated with newly created test suites and real-world apps compared with TaintDroid, CTT, and FlowDroid. The results demonstrate that VTDroid tracks more information flows resulting from the ATA techniques and generates fewer FPs than CTT.

Chapter 4 The community needs a reliable taint tracker for analyzing real-world apps. Researchers recently tested popular static taint analyzers and concluded that the tools are inaccurate and cannot be used for analyzing real-world apps dependably. On the other hand, researchers examined a famous dynamic taint tracker, TaintDroid, and pointed out that TaintDroid is the most difficult to set up compared to the static analysis tools they audited. Also, TaintDroid depends on specific devices and versions of Android OS released in 2013, narrowing down the scope of analyzable apps. Other dynamic analyzers are not effortlessly usable.

Chapter 4 proposes T-Recs, a runtime-data-utilized taint tracker that solves the current situation of no tracker that can analyze apps reliably. It records and reconstructs the app execution and performs taint analysis on an ordinary computer (e.g., a computer running Linux), not depending on Android OS. T-Recs' accuracy, analysis time, and success rate are evaluated in privacy leak detection compared to currently available taint analyzers, which are FlowDroid (w/ and w/o IC3), Amandroid, DroidSafe, DroidRA, IccTA, and TaintDroid (w/ and w/o IntelliDroid). The evaluation involves 158 test cases in DroidBench, 254 popular apps from the Google Play Store in 2016 and 2021, and 39,480 SDK-version-varied apps from the Google Play Store and Anzhi. The results show that T-Recs outperforms the compared tools in detection accuracy. T-Recs also achieves reasonable analysis time, app-runtime overhead, and success rate. VTDroid and T-Recs have been made available to the community.

Chapter 5 Chapter 5 concludes this thesis by summarizing each chapter. The regulations, market policies, and Android's data protection mechanisms should continue to be reformed in the future, and researchers should keep examining apps and libraries. VTDroid and T-Recs should be promising tools that enhance researchers' ability to analyze apps in the future.

Acknowledgements

I would first like to thank my supervisor, Professor Shoichi Saito, for all of his support throughout my undergraduate and graduate school career. I would also like to thank my supporting supervisor, Assistant Professor Shohei Kakei, for his valuable suggestions and discussions. Thanks are also due to Professor Koichi Mouri from Ritsumeikan University and Associate Professor Eiji Takimoto from Nara Women's University, who gave me much valuable advice. I would like to thank my parents, Yoshinobu Inayoshi and Kasumi Inayoshi, and all my family members for their support throughout the decade. They have always encouraged me and also provided the financial support I needed.

I gratefully acknowledge my Ph.D. supervisory committee, Professor Hiroshi Matsuo and Associate Professor Takahiro Uchiya from Nagoya Institute of Technology and Professor Atsushi Yoshida from Nanzan University, for their insightful feedback. I would further like to thank Professor Tomoaki Tsumura, Associate Professor Ryota Kawashima, Assistant Professor Shinsuke Kajioka, and all other faculty members of the Department of Computer Science at Nagoya Institute of Technology for their very helpful feedback and suggestions.

I am grateful to other students in the Saito and Kakei laboratory, the Matsuo and Kawashima laboratory, and the Tsumura laboratory for their valuable discussions. I thank Ryota Sakurai, Tatsuya Joho, Michinori Ohnishi, Kazuya Aoki, Yuka Hayashi, Kenta Nishimura, and Moeka Yamamoto for all of their support and encouragement. I would also like to thank Helton A. Yawovi, who will graduate in the same year, for his kind support. In addition, I thank the anonymous reviewers for their comments, which contributed to largely improving my papers. Finally, I would like to thank the Rinnai Scholarship Foundation for their financial support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Android OS started with version 1.0, released in 2008, and currently has 70% of the mobile OS market share in 2023 [1]. Android OS became the most popular mobile OS in the world. Users install Android applications (apps for short) to their Android devices. Apps are distributed through the official market, called Google Play Store [2], and other non-official markets, such as Aptoide [3] and F-Droid [4]. The Google Play Store currently provides 2.6 million apps [5]. There are 34 categories except for games, for example, communication, education, entertainment, finance, medical, health & fitness, and shopping. Such a wide variety of apps are downloaded and used in users' day-to-day activities on their devices, which are always connected to the internet.

With such smartphone development, the need to protect user privacy has increased. Regulations related to data protection have been put into operation in the past two decades. For example, the Children's Online Privacy Protection Act (COPPA) came into force in the US in 2000; the Personal Data Protection Act (PDPA) in Singapore in 2013; the California Consumer Privacy Act (CCPA) in CA, US in 2018; the General Data Protection Regulation (GDPR) in EU in 2018; the General Personal Data Protection Law (LGPD) in Brazil in 2020; and the Personal Data Protection Act (PDPA) in Thailand in 2022. Recently, mobile app developers have been fined. For example, the French regulator CNIL imposed a fine of 3 million euros on Voodoo, a smartphone game development company, in December 2022 [6]. The company used identifiers for advertising without user consent, which violated the French Data Protection Act (FDPA). According to the U.S. FTC announcement in May 2023 [7], an ovulation tracking app, Premom agreed to pay a fine of 100,000 USD for sharing users' health data with a third party and failing to notify the users of the incident. The Health Breach Notification Rule requires such notification to users.

Google has also made changes to restricting data access in the Android OS and policies in the Google Play Store. For example, accessing hardware information, which is unresettable and can be used for long-term tracking, has been restricted since Android 8 (2017) [8]. Specifically, they are settings and system properties, including boot date (ro.runtime.firstboot), camera's serial number (htc.camera.sensor.front_SN), Bluetooth MAC address property (persist.service.bdroid.bdaddr), and Bluetooth MAC address (Settings.Secure.bluetooth_address). Further, since Android 10, released in 2019 [9], apps installed from the Google Play Store are prohibited from accessing IMEI, MEID, ICCID, IMSI, device serial number, and MAC address. Also, privacy policies and privacy labels have become mandatory for all app developers since July 2022. The privacy label is also called the data safety section in the Google Play Store. The data safety section provides users with information about what data the app collects and what data the app shares with third parties. App developers are required to submit the information to the Google Play Store. Google states that the data safety section is the sole responsibility of the app developer.

1.1 Uncovering Privacy Protection Failures in Reality

While these regulations, market policies, and precautionary measures have been introduced, it is also essential to uncover how well app developers and third-party SDK providers follow the rules to protect user privacy. Security, privacy, and software engineering researchers have investigated real-world apps and found a large number of issues related to non-compliance, policy-violating, and protection-mechanism-circumventing behaviors in them.

Reyes et al. [10] investigated apps' compliance with COPPA and found violations in 19% of the apps they analyzed. Also, privacy policy violations have been uncovered in apps [11, 12, 13] and third-party libraries [14]. Andow et al. [15] developed PoliCheck, performing entity-sensitive analysis to detect inconsistencies between actual information leaks and privacy policies. Zhang et al. [16] investigated misconfigurations in analytics services and compared the actual behaviors of apps with the terms of service of analytics services. After the data safety section was introduced to the Google Play Store in 2022, Khandelwal et al. [17] uncovered that app developers are failing to disclose consistent information in the section.

Also, leaks of privacy-sensitive information have been detected [18]. DialDroid [19] is developed based on FlowDroid and detects data leaks caused by inter-app communications. Grace et al. [20] also analyzed privacy-sensitive information leaks in advertisements in apps. Agrigento [21] performed differential analysis and uncovered that popular apps and libraries apply complex combinations of encoding and encryption mechanisms to data being leaked.

Reardon et al. [22] detected apps that bypass the permission mechanism and leak sensitive information. Also, Meng et al. [23] found flaws that enable apps to obtain user-unresettable identifiers, which should not be accessed. Zhao et al. [24] uncovered backdoor and blocklist secrets in a large number of apps published on the Google Play Store and Baidu Market and pre-installed apps. Also, TriggerScope [25] uncovered suspicious triggers, such as backdoors in apps published on the Google Play Store.

Apposcopy [26] detects Android malware, and Gallingani et al. [27] detects vulnerabilities of inter-component communication in apps. WeChecker [28] is designed to uncover privilege escalation vulnerabilities.

It is expected that the regulations, market policies, and protection mechanisms will be changed in the future, and researchers should keep investigating real-world apps and libraries. App analysis techniques are fundamental to such investigation and must be upgraded continuously.

1.2 Existing Approaches to Mobile App Analysis

App analysis techniques have been developed and utilized by security, privacy, and software engineering researchers to uncover issues related to non-compliance, policy-violating, or protection-circumventing behaviors of apps. In terms of privacy, researchers often conduct the detection of leaks of privacy-sensitive information. Two main analysis technique options are taint analysis and network traffic analysis. Although network traffic analysis is widely used [12, 21, 18, 10, 22, 13, 15], this paper focuses on taint analysis. Taint analysis has the advantage of robustness against data transformation. In addition, taint analysis can be used for information leak detection and other tasks requiring information flow detection.

Static taint analysis requires no Android device and processes apps without running them. Since static taint analysis is scalable, it is popular for analyzing Android apps. Static taint trackers are widely utilized to detect privacy policy violations [11, 16, 14], privacy sensitive information leaks [19, 20], Android malware [26], vulnerabilities [27, 28], and other suspicious behaviors (e.g., backdoors) [24, 25].

Various tools of static taint analysis have been developed for app analysis. FlowDroid [29] is one of the most popular static taint analyzers and was developed in 2014. Also, DroidSafe [30] and Amandroid [31] are popular and general data flow analysis frameworks. IccTA [32], IC3 [33], and RAICC [34] are tools to precisely handle inter-component communication (ICC), one of the Android-specific features. Klieber et al. [35] and Bosu et al. [19] developed analysis techniques for tracking information flows through inter-app ICC. EdgeMiner [36] is designed to handle callbacks for tracking implicit control flows caused by the Android framework. Summarizing the Android framework to determine source and sink APIs is also challenging, and some approaches [37, 38] have been proposed. Another challenge is to handle Java-specific features, such as reflective calls (i.e., reflection), tackled by Barros et al. [39] and DroidRA [40, 41]. Furthermore, the accuracy of static taint analysis has been further improved with flow classification [42, 43, 44, 45, 46], dynamic heap snapshots [47], or path constraint computation [48]. DroidInfer [49] achieves higher scalability than FlowDroid by reducing resource consumption. More recently, static analysis techniques targeting native code have been developed, such as JN-SAF [50], CTAN [51], JuCify [52], and μDep [53].

On the other hand, many dynamic taint trackers have been developed to detect privacy-sensitive information leaks in apps or analyze mobile malware. TaintDroid [54] is a well-established tool for dynamic taint analysis for Android apps. Since dynamic taint trackers only analyze executed paths, it is highly accurate. In its evaluation, TaintDroid generates no FPs in privacy leak detection with 30 popular real-world apps. Tripp et al. [55] developed BayesDroid based on TaintDroid to improve its accuracy further. Wei et al. [56] proposed LazyTainter, which enhances TaintDroid's memory efficiency. AppsPlayground [57] is a performance-oriented dynamic analysis system utilizing TaintDroid for large-scale app analysis. VetDroid [58] focuses on app behaviors related to the usage of sensitive system resources and utilizes a dynamic taint analysis to track such resources.

More recently, researchers have developed hybrid analysis techniques utilizing static taint analysis to assist a dynamic taint analysis. IntelliDroid [59] performs targeted execution, enabling a dynamic taint tracker to run a specific code path. Similarly, Harvester [60] can improve a dynamic taint tracker by triggering malicious code.

Since Android version 5.0, released in 2014, a new Android runtime environment called Android Runtime (ART) [61] has been introduced to replace the previously used runtime called Dalvik Virtual Machine (VM). While the Dalvik VM employs an interpreter that performs just-in-time compilation, the ART employs ahead-of-time (AOT) compilation, improving performance and battery life. Since TaintDroid relies on the Dalvik VM, it cannot be used on Android version 5.0 and later and cannot analyze apps with minimum SDK versions equal to or higher than Android version 5.0. TaintART [62], ARTist [63], and TaintMan [64] were proposed as dynamic taint trackers compatible with the ART. There is also a dynamic taint tracker [65] that instruments the target app's code to inject a taint logic. The dynamic taint analysis tools mentioned so far are bytecode-level taint trackers, only analyzing apps' bytecode (i.e., DEX bytecode [66]). The DEX bytecode format maintains variable semantics, helping a taint tracker to distinguish between data references and scalar values. When an instruction operates on data, the data is always read into a virtual register.

Unlike the bytecode-level taint trackers, there are also native-level taint trackers. DroidScope [67], NDroid [68, 69], and Malton [70] analyze apps' native code to track information flows more comprehensively.

1.2.1 ATA Problem

Cavallaro et al. [71] discuss anti-taint-analysis (ATA) techniques allowing adversaries to circumvent binary-level taint trackers. Pointer indirection, control dependence, and timing channels can be used to cause information flows that are difficult to track. More recently, Sarwar et al. [72] demonstrate a series of ATA techniques on the Android platform. An app can conduct one of the techniques to transfer information without triggering taint propagation of the bytecode-level taint trackers on the Android platform. They present control dependence, which exploits control flows, and the subversion of benign code, which abuses commands of the Linux system. They also define side channels as techniques exploiting any medium that can represent information. They implemented 16 techniques into a proof of concept called ScrubDroid [73]. The techniques are only a few lines of code each and are shown to be sufficient to bypass TaintDroid, one of the most popular taint analysis tools, completely. Similar techniques were discovered in some real-world apps [74]. In addition, Tigress, a program obfuscation tool, presents ATA as one of its transformation methods [75]. Tigress' ATA implementation is based on works by Sarwar et al. [72] and Cavallaro et al. [71]. Such a tool raises a concern that the well-known ATA techniques from ScrubDroid can be injected into any benign apps on app stores by their developers to defend them against a taint analysis. Consequently, Android security and privacy studies that utilize current taint trackers must produce unreliable results.

Most of the work on taint analysis for Android apps states that ATA techniques are out-of-scope. Although there are a few approaches to tracking information flows against ATA techniques, such as [76], they are only partially effective against ATA techniques and are still vulnerable to the well-known techniques presented by ScrubDroid.

Another problem is that the classification in ScrubDroid does not effectively reveal features of flows caused by the techniques, making it challenging to discuss which techniques can be handled by what type of taint trackers, such as data-flow trackers and control-flow trackers.

1.2.2 Accuracy and Usability Problems

Static taint analysis has the problem of detecting incorrect execution paths, increasing the cost of verifying experiment results. Recent reviews of the literature on static taint analysis showed the limitations of the analysis [77, 78]. Zhang et al. evaluated currently-available static taint analysis tools: FlowDroid, Amandroid [31], and DroidSafe [30] with a test suite called DroidBench [79] and real-world apps. The results show that the tools are inaccurate and cannot be used for analyzing real-world apps dependably. The increase in false positives (FPs) complicates analysis-result verification, causing an increase in analysis cost. For example, Zhao et al. manually analyzed 70 out of over 16,000 detected apps to estimate the accuracy, and the result is 87.14% (i.e., nine apps are FPs) [24]. Three of the FPs were caused by conflicting constraints along the execution path. Such a manual analysis, especially finding path constraints, is complex and requires significant effort. Also, increasing the complexity of the analysis algorithms increases the analysis time, preventing the analysis from completing in a reasonable time.

On the other hand, a dynamic taint analysis uses the target app's runtime semantics and only analyzes the executed paths. There is no chance of detecting incorrect execution paths. TaintDroid is one of the most popular dynamic taint analysis tools and is publicly available. However, Reaves et al. [80] discuss that TaintDroid is the most difficult to set up in comparison with static analysis tools they audited because TaintDroid requires the user to build the Android from the source code. Also, TaintDroid depends on specific devices and versions of Android OS, and a supported device is not always available. It also narrows down the scope of analyzable apps. There are more tools [62, 63, 65, 64], which perform the bytecode-level dynamic taint analysis for Android apps other than TaintDroid. Although, in comparison with static taint analyzers, dynamic taint trackers have been barely reviewed in the community except for TaintDroid. They are not effortlessly usable.

Another drawback in current dynamic taint trackers is that the app exercise needs to be executed every time the taint analysis runs because the app exercise and the taint analysis are performed simultaneously in TaintDroid. Therefore, when the analyst changes the parameters of the taint analysis (e.g., the data to be tracked) and re-analyzes the same app, the app exercise also needs to be repeated, which incurs extra costs. It also distresses researchers who add and evaluate new features to the taint analysis.

1.3 Taint Analysis with Runtime Data

This paper discusses the utilization of the app's runtime data to improve taint analysis for Android apps. Specifically, this paper proposes two approaches named VTDroid [81] (Chapter 3) and T-Recs [82] (Chapter 4). VTDroid is designed to track information flows against ATA techniques. T-Recs is developed to detect information flows accurately while reducing the dependency on Android OS.

VTDroid focuses on the well-known ATA techniques shown in ScrubDroid, which might be introduced into apps with obfuscator tools by app developers to protect their apps against the analysis. So far, there is no tracker that can handle the combinations of the ATA techniques. VTDroid's goal is to make it difficult for apps to evade the taint tracking by neutralizing such recognized and uncomplicated techniques (i.e., ScrubDroid) not specific to a particular technique. VTDroid is based on a characterization of information flows. Information can flow on the runtime layer and across API calls by data flows, resulting from assignment instructions, and other channels that are control dependence, pointer indirection, and timing channels [71]. This paper takes them into consideration and characterizes the ATA techniques by four types of information flow. This paper presents a new taint-tracking technique, implemented into VTDroid, that detects the four types of information flow even if they occur through unmonitored areas via API calls. In order to reduce the dependency on the API method list, which is a disadvantage of current tracker [76], the approach performs value logging and matching that propagate taint among registers based on their data values, in addition to the traditional bytecode-level tracking. VTDroid also employs information-preservability inspection to reduce the amount of FPs.

T-Recs is a new runtime-data-utilized taint tracker that solves the current situation of no taint tracker that can reliably analyze real-world apps, especially recently released apps. T-Recs records and reconstructs the app execution, and taint analysis is performed not on an Android device but on any computer, including a desktop or laptop. T-Recs' implementation does not depend on Android OS and can avoid the TaintDroid's usability issue. Further, with T-Recs, the analyst can start analyzing apps immediately after plugging an unmodified device into their computer. T-Recs consists of five components: parser, instrumentator, logger, reconstructor, and exerciser. First, the parser and instrumentator run on a computer and modify the target app to inject the logger. Then, the modified app is installed and launched on an Android device, and the logger also starts running and recording the app's runtime data at almost instruction by instruction. Finally, on a computer, the reconstructor reproduces the app execution based on the parsed and logged data, and taint analysis is executed along with it. Since the reconstructor is independent of the actual app execution, the taint analysis can be re-executed without exercising the app. The last component, the exerciser, addresses how to trigger the target behavior in apps, a general challenge in dynamic analyses. Since a dynamic taint analysis only analyzes the executed part of the app's code, triggering the target behavior in the app is necessary. However, app exercise depends on the apps and the data to be tracked, and it is not trivial. For example, tracking user inputs requires input-related exercises. In this paper, the exerciser is explicitly designed for DroidBench, a popular test suite for evaluating taint analysis tools. The exerciser automatically triggers ICC, callbacks, and lifecycle events in DroidBench apps so that T-Recs can be evaluated compared to static taint trackers, which have no coverage issues.

1.4 Contributions

In Chapter 3, towards tracking information flows caused by ATA techniques, first, all the 16 techniques in ScrubDroid are characterized by newly defined types of information flow (Section 3.2). New test suites are created and contain 31 ATA techniques against privacy leak detection and 28 ATA techniques against the validation detection. Based on the characteristics of the information flows, a unified method called value-utilized taint propagation is devised to avoid FNs against the multiple ATA techniques that are not specific to a particular topic (Section 3.3). The method is implemented into a bytecode-level tracker called VTDroid for app analysis (Section 3.4). VTDroid tracks data flows and the other types of flow altogether. The effectiveness of VTDroid is evaluated with a test suite and real-world apps collected from the Google Play Store and Baidu app store (Section 3.5). VTDroid is compared to TaintDroid, CTT, and FlowDroid for privacy leak detection and is also evaluated by being used as the detector of user input validation for InputScope. The results demonstrate that VTDroid produces smaller FNs than current trackers, TaintDroid, CTT, and FlowDroid, while generating fewer FPs in privacy leak detection than the current solution, CTT. In validation detection, VTDroid tracks more information flows resulting from the ATA techniques while generating slightly more false positives than FlowDroid, a default tracker in InputScope. Also, the VTDroid's verification cost of FPs is negligible.

Chapter 4 explains the app-runtime recording and reconstruction mechanism (Section 4.2), implemented into a new taint analysis system called T-Recs with nearly 17,000 lines of

Python and Smali code (Section 4.3). T-Recs' accuracy, analysis time, and success rate were evaluated in privacy leak detection compared to currently available taint analyzers, which are FlowDroid (w/ and w/o IC3), Amandroid, DroidSafe, DroidRA [40, 41], IccTA, and TaintDroid (w/ and w/o IntelliDroid [59]) (Section 4.4). The evaluation involves DroidBench, 254 popular apps from the Google Play Store in 2016 and 2021, and SDK-version-varied apps from the Google Play Store and Anzhi [83]. The results show that T-Recs outperforms the compared tools in detection accuracy. T-Recs also achieves reasonable analysis time and success rate. T-Recs' app-runtime overhead (i.e., the overhead for apps to be installed, be launched, cause leaks, and be uninstalled) and parallel execution performance were also evaluated in comparison with the other trackers. The results are acceptable, and running T-Recs in parallel can easily shorten the analysis time. An additional experiment shows that the importance of tracking ICC- and reflection-related flows is highlighted by T-Recs, detecting leaks related to ICC and reflection, missed by FlowDroid in popular apps collected from the Google Play Store in 2016 and 2021. A debugging feature was added to the reconstructor to identify and count the leaks. Then, only the reconstructor was executed. This experiment indicates that T-Recs' cost of re-executing taint analysis is small, taking 34 minutes (17% of the whole) for the 96 apps collected in 2016 and one hour and 40 minutes (11% of the total) for the 158 apps collected in 2021. T-Recs has been made available to the community.

Chapter 2

Background

This chapter first explains how information flows occur in general program code. Then, it describes Android-specific and Java-specific features that are keys to Android app analysis. Lastly, it discusses the taint analysis fundamentals and current taint trackers for Android app analysis.

2.1 Information Flow

This section explains a variety of information flows. Information flow can be considered a dependency between variables where one holds a value derived from the other's value [84]. When the value of y changes depending on the value of x, there is information flow from x to y. This section describes four types of information flows: direct assignment, memory operation, control dependence, and timing channel, which are discussed in the literature on ATA techniques [71].

2.1.1 Direct Assignment

This type of information flow is caused by assignment instructions, such as =. Figure 2.1 shows an example of direct assignment information flow. The x's value is moved to y by = instruction, and y will have the same value as x's. There is a direct dependency between the two variables, and there is information flow from x to y. This type of information flow is commonly called data flow.

Since it is obvious that the right side of = is the source of data flow and the left side is the destination, tracking this information flow is straightforward. An information flow tracker can track this flow as long as the tracker can monitor the instruction.

2.1.2 Memory Operation

This type of information flow is caused by correlating the information with memory addresses. Figure 2.2 shows an example of such information flow caused by an array. Lines 1 and 2 are code for creating a table where each element is the same value as its index. In this code, *table* has 256 values, which can be extended easily. Information flow from x to y is caused at Line 3. The value of x is used as an index to obtain a value in *table*, and the value is assigned to y. As a result, x and y hold the same value, and the information can be preserved. Note that x's value is not directly assigned to y.

```
y = x;
```

FIGURE 2.1: Information flow $x \Rightarrow y$ by direct assignment.

```
for (int i = 0; i < 256; i++) {
    table[i] = i; }
y = table[x];
```

FIGURE 2.2: Information flow $x \Rightarrow y$ by pointer indirection.

An information flow tracker can detect this information flow if the instruction can be monitored. Specifically, the index and destination operands should be clear when the instruction is processed.

2.1.3 Control Dependence

A control flow causes this type of information flow. A control flow is a branching behavior of a program. It represents the order of execution of instructions that may occur during program execution. This type of information flow is commonly called implicit flow.

Figure 2.3 shows a simple code snippet that causes this type of information flow from x to y. There is an if-else statement, and the condition holds if x's value is 0. Assume that x's value is always either 0 or 1, y is assigned 0 if x's value is 0, and y is assigned 1 if x's value is 1. Although there are no assignment statements between the two variables, y will be the same value as x's, and the information is preserved. Even if x has a broader range of values, the code snippet can be easily extended and can transfer arbitrary values.

An information flow tracker should detect control-dependent statements to track this information flow. One approach is program counter tainting [85, 42]. If an operand of the if statement is tainted, the program counter is also tainted. Then, if the program counter is tainted, the destination operand of each instruction is also tainted. Further, You et al. [64] developed a technique that checks the values of variables at re-convergence points to identify if the information is preserved, and the accuracy would be improved. This technique is effective as long as the destination of information flow can be monitored at the re-convergence points.

2.1.4 Timing Channel

This type of information flow is caused by encoding data into timing. A function *sleep()* or control instruction is used to delay the program execution, and the delay is measured to extract the information. Figure 2.4 shows a simple example of a timing channel using *sleep()*. The delay changes depend on the x's value in Line 2, and the delay is measured and saved into y in Lines 1 and 3. Since the delay can preserve the information in x, there is information flow from x to y. The delay can be measured outside the process (i.e., the receiving side) [71] so that *time()* function is not necessary.

An information flow tracker can detect this information flow if the tracker knows what instructions can delay the program execution. Also, the tracker should be able to monitor the execution of such instructions. Researchers developed techniques that detect timing channels by analyzing API usage frequencies [86] or the time differences between send operations [87].

```
if (x == 0) {
    y = 0;
} else {
    y = 1; }
```

FIGURE 2.3: Information flow $x \Rightarrow y$ by control dependency.

```
start = time();
sleep(x);
y = time() - start;
```

FIGURE 2.4: Information flow $x \Rightarrow y$ by timing channel.

2.2 Android App Analysis Key Features

This section explains Android-specific and Java-specific features that are keys to information flow analysis of Android apps.

2.2.1 Android-Specific Features

Android-specific features are challenges in Android app analysis. An app has no *main()* function that is usually an entry point of a program and instead has callback functions that are triggered by system and user events. Therefore, the Android platform itself or its model is necessary to analyze apps; otherwise, the execution order of functions in an app is unknown. In addition, callback functions can be dynamically registered, and static analysis would be ineffective in such cases. The rest of this section explains major Android-specific features.

Inter Component Communication

Inter-component communication is the communication between components of apps. An app usually has some components called activities, and each activity provides a user interface of specific tasks to the user. An activity can launch another activity and receive the result from it. In the communication, a message called Intent indicates the target activity name and data passed to the activity. If the target activity name is specified in an intent, it is called explicit Intent, and if the name is not specified, it is called implicit Intent. Intent messages are handled by intent filters based on component name, category, MIME type, and the set of action strings. They are strings and are necessary for analysis tools to identify who receives what intent message. However, such strings can be loaded from the network or files and can also be encrypted or obfuscated, and correctly matching the intent messages to intent filters is challenging [80].

Figure 2.5 shows an example of ICC, which is ActivityCommunication6 from DroidBench. There are two activities: *OutFlowActivity* and *InFlowActivity*. *OutFlowActivity.onCreate()* loads privacy-sensitive data and sends an intent message to launch *InFlowActivity*. *InFlowActivity.onCreate()* receives the message and writes the data to the log. An information flow tracker should detect this control flow. In addition, the tracker must detect the data flow between the activities via the intent message. The data is stored in the intent message by *putExtra()* in *OutFlowActivity.onCreate()* and obtained by *getStringExtra()* in *InFlowActivity.onCreate()*. A field called extras is used to keep the data. The field is a key-value storage

```java
public class OutFlowActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TelephonyManager telephonyManager =
            (TelephonyManager) getSystemService(
                Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source

        Intent i = new Intent(this, InFlowActivity.class);
        i.putExtra("DroidBench", imei);

        startActivity(i);
    }
}

public class InFlowActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent i = getIntent();
        String imei = i.getStringExtra("DroidBench");
        Log.i("DroidBench", imei);  // sink
    }
}
```

FIGURE 2.5: Example code snippet of ICC.

structure, and the key is “DroidBench” in this example. The tracker should collect the key to differentiate values when multiple values are stored in the extras.

Lifecycle

Lifecycle indicates a series of states of an app. For example, when the user launches an app, the app’s state is “created”, and when the user taps the home button and sends the app to the background, the app’s state is “paused”. The app can perform appropriate procedures depending on its state to avoid unnecessary resource usage. In addition, when the user returns to the app, the app can let the user start using it continuously from the previous point the user left the app.

More specifically, an app consists of four components: activities, services, broadcast receivers, and content providers. Activities provide user interfaces and have the most complex lifecycle. The Activity class provides a core set of six callbacks: *onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()*, and *onDestroy()*. The Android system invokes each of these callbacks as the activity enters a new state. An information flow tracker should consider the lifecycle callbacks to track control flows. Each component is statically defined so that the tracker can detect the flow; however, there is an exception that the broadcast receiver component is dynamically created, and program analysis tools often overlook it [80].

```java
public class MainActivity extends Activity {
    public static final String KEY = "DroidBench";

    @Override
  public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        if (savedInstanceState != null) {
            String value = savedInstanceState.getString(KEY);
            Log.i("DroidBench", value);  // sink
        }
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState)
    {
        TelephonyManager mgr =
            (TelephonyManager) this.getSystemService(
                TELEPHONY_SERVICE);
        String imei = mgr.getDeviceId();  // source

        savedInstanceState.putString(KEY, imei);

        super.onSaveInstanceState(savedInstanceState);
    }
}
```

FIGURE 2.6: Example code snippet of lifecycle.

Figure 2.6 shows an example of information flow caused by lifecycle callbacks. The code is from ActivitySaved1 in DroidBench. There is an activity called *MainActivity*. When the user taps the app's icon on the home screen and the app is launched, the activity is created, and *MainActivity.onCreate()* is invoked. Then, the user presses the home button, *MainActivity.onSaveInstanceState()* is invoked, and privacy-sensitive data is obtained and stored into *savedInstanceState*. Subsequently, if the user does not return to the app for a while, the app is killed, and the activity is destroyed. When the user relaunches the app, since the activity has been destroyed, *MainActivity.onCreate()* is invoked again. In the method, the privacy-sensitive data is retrieved from *savedInstanceState* and is written to the log. An information flow tracker must understand the sequence of invocations of *MainActivity.onCreate()* and *MainActivity.onSaveInstanceState()* as well as the data flow via *savedInstanceState* to detect the information flow.

Callbacks

In addition to lifecycle callbacks, there are more, such as system and UI callbacks. System callbacks are invoked at system events. For example, *onLocationChanged()* method is called by the Location Manager Service when the device geolocation is changed. UI callbacks are triggered depending on user-interface events, such as button taps by the user. An information flow tracker should consider these callbacks as the app's entry points. Additionally, an app can contain a custom widget, which makes the analysis more complex [80].

```java
public class Button1 extends Activity {
    private static String imei = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_button1);

        TelephonyManager telephonyManager =
            (TelephonyManager) getSystemService(
                Context.TELEPHONY_SERVICE);
        imei = telephonyManager.getDeviceId();  // source
    }

    public void sendMessage(View view){
        Log.i("DroidBench",
            ((Button)view).getHint().toString());  // sink
        ((Button)view).setHint(imei);
    }
}
```

FIGURE 2.7: Example code snippet of UI callbacks.

```xml
<!-- activity_button1.xml -->
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".Button1" >
    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="185dp"
        android:text="@string/button"
        android:hint="@string/button"
        android:onClick="sendMessage"/>
</RelativeLayout>
```

FIGURE 2.8: XML file content corresponding to the example code snippet of UI callbacks.

```
Class<?> clz = Class.forName(getString(R.string.class_name));
Method method = clz.getMethod(getString(R.string.method_name));

Class<?> clz2 = method.getDeclaringClass();
BaseClass bc = (BaseClass) clz2.newInstance();
```

FIGURE 2.9: Example code snippet of reflection.

```
<!-- strings.xml -->
<string name="class_name">edu.wayne.cs.ConcreteClass</string>
<string name="method_name">foo</string>
```

FIGURE 2.10: XML file content corresponding to the example code snippet of reflection.

Figure 2.7 shows a code snippet of *Button1* activity, which is a UI callback, and Figure 2.8 presents a part of the corresponding XML file (activity_button1.xml). They are obtained from the Button5 test case in DroidBench. When the app is launched, *Button1.onCreate()* is invoked, and privacy-sensitive data is obtained and stored in the static field. As the snippet from the XML file (Figure 2.8) shows, there is a button with *sendMessage()* as its callback method, configured at Line 17. In *sendMessage*, first, the data is obtained by *getHint()* and is written to the log. Then, data in the static field is passed to *setHint()*. These lines of code indicate that the button should be tapped twice to cause information flow of the sensitive data. An information flow tracker should detect such multiple callback invocations as well as the data flow through the static field to track the information flow.

2.2.2 Java-Specific Features

There are Java-specific features that complicate the app analysis, as well as Android-specific features. Java code relies heavily on a class hierarchy structure, making it more challenging to detect actually-executed methods and fields without running the code [80]. Besides that, this section explains two major language features: reflection and exception.

Reflection

Reflection enables an app developer to resolve a method to be invoked or a field to be accessed dynamically. Using reflection can obfuscate apps, but many apps use reflection for other purposes as well [88]. Therefore, entirely prohibiting reflection usage is not acceptable. A target class, method, or field name is provided as a string. An app developer can obfuscate the string until right before a reflection call, making a static analysis harder. An information flow tracker must handle reflective invocations to track control flows.

Figure 2.9 shows an example of a reflective call in the Reflection7 test case from DroidBench. Strings of class and method names are defined in the XML file (Figure 2.10) and can be accessed by *R.string.class_name* and *R.string.method_name*. First, the Class object is created by *forName()* with *R.string.class_name* as an argument. Then, the Method object is generated by *getMethod()* with *R.string.method_name* as an argument. The Method object's class is obtained by *getDeclaringClass()*, and the class is instantiated by *newInstance()*. These lines of code can hide the actual class and method names, making the app analysis more difficult.

```
public class Exceptions3 extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_exceptions3);

        String imei = "";
        try {
            TelephonyManager telephonyManager =
                (TelephonyManager) getSystemService(
                    Context.TELEPHONY_SERVICE);
            imei = telephonyManager.getDeviceId();  // source
            int[] arr = new int[42];
            if (arr[32] > 0)
                imei = "";
        }
        catch (RuntimeException ex) {
            SmsManager sm = SmsManager.getDefault();
            sm.sendTextMessage(
                "+49 1234", null, imei, null, null);  // sink
        }
    }
}
```

FIGURE 2.11: Example code snippet of exception.

Exception

Exception can complicate the control flow of a program, making it more difficult to track information flows. A challenge is determining whether an exception occurs at lines of code. If any exception never occurs in a try block, the corresponding catch block will never be executed, and tracking information flows through the catch block can lead to false positives. It has been pointed out that tracking exceptional flows can result in a substantial amount of false positives [89].

Figure 2.11 shows a code snippet with a try-catch statement from Exceptions3 in DroidBench. Privacy-sensitive data is loaded in the try block and is sent via an SMS message in the catch block. The catch block is executed when *RuntimeException* occurs in the try block, and the flow of the sensitive data is caused. However, in this example, the catch block will never be executed because *RuntimeException* will never occur in the try block. If an information flow tracker falsely assumes that the catch block is executed, the information flow is falsely detected (i.e., false positive).

2.2.3 Other Features

There are more features specific to Android or Java. Android enables apps to save their data to files and databases on storage such as internal storage and SD cards. Therefore, an information flow tracker should track dataflows through such storage. Automatically selecting sources and sinks for information flow analysis is also challenging. However, this paper considers these features out-of-scope.

Also, there are Java-specific features such as dynamic code loading and native code. Dynamic code loading is often overlooked in the analysis of Java code [80]. Java supports the execution of native code. Analyzing native code requires completely different techniques from Java analysis. Native code is often ignored by the analysis tools as well as dynamic code loading [80].

2.3 Taint Analysis of Android Apps

This section explains taint analysis, especially for Android app analysis. First, it describes the fundamentals of taint analysis. Then, it introduces current taint trackers for Android app analysis.

2.3.1 Taint Analysis Fundamentals

In this paper, taint analysis is considered to be a technique to detect information flows. Taint tracking and information flow tracking can be considered different [71]; however, they are considered the same in this paper. An information flow from a register x to another register y, denoted $x \Rightarrow y$, is caused by an operation, or series of operations, that uses the value of x to derive a value for y [84]. Tracking information flows can reveal the program's behavior. For example, an analyst can understand what information is sent to the network by the program.

In taint analysis, information that an analyst needs to track is marked with a tag called a taint tag. Taint tags are propagated along with the information flows and are checked at specific points in the program. Taint analysis consists of three operations: introduction, propagation, and checking.

Taint Introduction

Taint introduction is an operation that assigns a new taint tag to a register or a variable. Before the analysis, the user must configure the execution conditions of the taint introduction. Such configuration includes instruction names and operand types and is called a taint source. For example, if the user tries to find leaks of phone numbers, the taint source is a register holding the return value of a function returning a phone number.

Taint Propagation

Taint propagation is an operation that assigns taint tags of a register to another register. It is usually performed at each instruction that causes information flows. As a result, all registers holding the information that should be tracked are assigned taint tags. For example, suppose an instruction copies the value of the source operand to the destination operand. In that case, there is a data flow, and the taint tag of the source operand is propagated to the destination operand to track the flow. If an instruction has multiple source operands, the taint tags of the source operands should be combined and propagated to the destination operand. These rules are called taint propagation rules.

Taint Checking

This operation checks whether a register has taint tags at pre-configured points in the program. Such points are called taint sinks. For example, suppose the user is trying to uncover information leaks and configures a tracker to check taint tags at functions sending network data. In that case, the tracker performs the taint checking at the functions and notifies the user if taint tags are found. Consequently, the user can understand whether information flows from the taint source to the network.

2.3.2 Current Taint Trackers for Android App Analysis

When analyzing apps from the Google Play Store or third-party markets, their source code is not usually available. Therefore, taint trackers should analyze apps at bytecode or binary levels. TaintDroid [54] is a well-established tool of dynamic taint analysis for apps, employing bytecode-level tracking that is the most common taint-tracking approach utilized by many systems [29, 63, 90, 65, 62, 64]. TaintDroid detects information flows by interpreting apps' instructions of the Dalvik executable (DEX) bytecode, operating with virtual registers (registers for short). TaintDroid marks registers as tainted based on taint propagation rules defined for each instruction, and a chain of tainted registers represents information flows.

Bytecode-level tracking is common because the DEX bytecode format preserves variable semantics, which enables a tracker to distinguish between pointers, which are references to data, and scalar values. Without variable semantics, a taint tracker can assign taint tags to pointers, and the taint tags would quickly spread by pointer dereferences. It is called taint explosion and can occur in native-level trackers. Although, there are native-level trackers such as Malton [70], NDroid [69, 68], and DroidScope [67]. They target both DEX bytecode and machine language instructions and are mainly aimed at analyzing Android malware.

Chapter 3

VTDroid: Value-utilized Tracking of Information Flows

The number of Android apps has rapidly increased in app stores, such as the Google Play Store. In order to detect apps' bugs and policy-violating behaviors, security researchers utilize app analysis techniques, which have been actively developed in the past decade. Taint analysis is widely used to uncover hidden features, for example, trigger-based behaviors [25] and policy inconsistencies [11]. Recently, researchers developed a tool called InputScope [24], revealing 12,706 backdoors and 4,028 censorship secrets in over 150,000 apps published on the Google Play Store and an unofficial store and pre-installed apps. InputScope utilizes a taint tracker to detect user input validations, which relate to the backdoors and censorship secrets.

TaintDroid [54] is a well-established tool of dynamic taint analysis for apps, employing bytecode-level tracking that is the most common taint-tracking approach utilized by many systems [29, 63, 90, 65, 62, 64]. TaintDroid detects information flows by interpreting apps' instructions of the Dalvik executable (DEX) bytecode, operating with virtual registers (registers for short). TaintDroid marks registers as tainted based on taint propagation rules defined for each instruction, and a chain of tainted registers represents information flows.

Since the taint propagation rules target only the bytecode instructions executed on the runtime layer of the Android platform, the trackers will suffer from false negatives (FNs) for apps transferring information across API calls. Information can flow from one register to another across API calls, hidden from TaintDroid. Sarwar et al. [72] demonstrate anti-taint-analysis (ATA) techniques with their proof-of-concept implementation called ScrubDroid [73] that exploit such obscured flows to hide their apps' behaviors from TaintDroid. Tigress, a program obfuscation tool, presents ATA as one of its transformation methods [75]. Tigress' ATA implementation is based on works done by Sarwar et al. [72] and Cavallaro et al. [71]. Such a tool raises a concern that the well-known ATA techniques from ScrubDroid can be injected into any benign apps on app stores by their developers to defend them against a taint analysis. Consequently, Android security and privacy studies that utilize current taint trackers must produce unreliable results.

With the motivation of tracking such information flows, Graa et al. devised a context-tainting tracker (CTT) [76]. CTT utilizes a hand-picked list of API methods causing information flows that TaintDroid overlooks. Their implementation lists nine specific methods in the Android API and the runtime, such as *setTextScaleX()* and *getTextScaleX()* of the TextView class, covering six ATA techniques in ScrubDroid. However, it is impractical to list all exploitable API methods among more than 110,000 API methods provided by the Android [37], and FNs will still be produced when unlisted methods are leveraged. CTT can also increase the number of false positives (FPs) because the listed methods are usually invoked for purposes other than information transfer. Furthermore, the list depends on the versions of the Android platform, and the list would be ineffective in the case that the Android platform is updated.

This chapter presents a new taint-tracking technique that detects information flows occurring across API calls [81]. In order to reduce the dependency on the API method list, the approach performs value logging and matching that propagate taint among registers based on their data values, in addition to the traditional bytecode-level tracking. The technique is based on the characterization of ATA techniques. Information can flow on the runtime layer and across API calls by data flows, resulting from assignment instructions, and other channels that are control dependence, pointer indirection, and timing channels [71]. Consequently, the ATA techniques can be characterized by four types of information flow.

Towards precise detection of ATA techniques' information flows by utilizing register values, this chapter focuses on evaluating the effectiveness and FP amount of exact value matching in detecting information flows across API calls. Previous studies [91, 64] developed techniques that inspect flows' information preservability by leveraging runtime data values to accurately detect information flows occurring with control dependence. It is expected that other types of flows could be detected by inspecting information preservability with data values, and the amount of FPs could be reduced. However, no study that explores the approach has been found. Therefore, as a first step, this chapter presents a technique for detecting information flows across API calls based on the simplest method of information-preservability inspection, which is exact value matching. This chapter would provide a lower bound of the effectiveness and the amount of FPs in the value-utilized detection for future development of more robust inspection methods.

The technique was implemented into a bytecode-level tracker called VTDroid for app analysis. VTDroid tracks data flows and the other types of flow altogether. The effectiveness of VTDroid was evaluated with a test suite and real-world apps collected from the Google Play Store and Baidu app store. VTDroid was compared to TaintDroid, CTT, and FlowDroid for privacy leak detection, and the results show that VTDroid outperforms the trackers. The apps were also analyzed with InputScope utilizing VTDroid as the detector of user input validations. The results demonstrate that VTDroid tracks more information flows resulting from the ATA techniques while generating slightly more false positives than FlowDroid, a default tracker in InputScope.

Here is the summary of the contributions:

- All the 16 techniques in ScrubDroid are characterized by defined types of information flow and created test suites containing 31 ATA techniques against privacy leak detection and 28 ATA techniques against the validation detection.
- Based on the characteristics of the information flows, a unified method called value-utilized taint propagation is devised to avoid FNs against the multiple ATA techniques that are not specific to a particular topic.
- VTDroid is developed, and it is demonstrated that VTDroid produces smaller FNs than current trackers, TaintDroid, CTT, and FlowDroid, while generating fewer FPs than the current solution, CTT, in privacy leak detection, and the VTDroid's verification cost of FPs is negligible in the validation detection.

The rest of this chapter is organized as follows. Current taint analysis and ATA techniques are further explained in Section 3.1. Characteristics of ATA techniques are discussed in Section 3.2. The approach is presented in Section 3.3, and its implementation, called VTDroid, is described in Section 3.4. The evaluation is reported in Section 3.5, and the capabilities of VTDroid are explained in Section 3.6. Related work is discussed in Section 3.7. Lastly, the summary is stated in Section 3.8.

3.1 Background

This section first explains current systems for Android app taint analysis. Then, this section discusses ATA techniques and countermeasures.

3.1.1 Taint Tracking for App Analysis

An information flow from a register x to another register y, denoted $x \Rightarrow y$, is caused by an operation, or series of operations, that uses the value of x to derive a value for y [84]. TaintDroid is a representative bytecode-level taint tracker in the Android platform and detects $x \Rightarrow y$ within an app by interpreting the app's bytecode instructions as follows. First, TaintDroid marks a register as tainted at some input point (i.e., taint source). Then, TaintDroid propagates taint from the register to other registers based on taint-propagation rules defined for each bytecode instruction. As a result, registers holding values derived from a taint source are marked as tainted. TaintDroid checks the taint at an output point (i.e., taint sink) and identifies information flows from the input to output points.

Overtainting and undertainting are errors occurring in taint analysis [92]. Overtainting means that a register is marked as tainted when its value is derived from none of the taint sources. It is a prolonged problem for binary-level trackers, which have been developed long before Android-targeting trackers. However, TaintDroid can suppress it with variable semantics in the DEX bytecode, which is the primary reason for tracking flows in apps at only the bytecode level. In contrast, undertainting means that a register is not marked as tainted (i.e., untainted) when its value is truly derived from a taint source. It leads to FNs that a malicious app is classified as benign or bugs are overlooked. One major cause of undertainting in the bytecode-level tracking is API methods, unmonitored by the previously-developed trackers [29, 63, 54, 90, 65, 62, 64]. TaintDroid uses profiles to propagate taint among parameters, class members, and return values of the Java Native Interface (JNI) methods to avoid missing such uncertain flows. However, it triggers FNs for JNI methods causing flows other than to the return value [54]. StubDroid [38] addresses the formalization of the taint rules for the tracking across API calls. StubDroid automatically generates summaries of data flows in the Android framework; however, summaries are manually created for native code, and FNs occur with methods for which summaries have not been created.

3.1.2 ATA Techniques and Countermeasures

Cavallaro et al. [71] discuss ATA techniques allowing adversaries to circumvent binary-level trackers, and Sarwar et al. [72] demonstrate a series of ATA techniques that an app conducts to transfer information without triggering taint propagation of the bytecode-level trackers on the Android. They present control dependence, which exploits control flows, and the subversion of benign code, which abuses commands of the Linux system. They also define side channels as techniques exploiting any medium that can represent information. They call the techniques side channels in the meaning of the channels that are out of the scope of the tracking. They implemented 16 techniques into a proof of concept called ScrubDroid. The techniques are only a few lines of code each and are shown to be sufficient to completely bypass TaintDroid. For example, the TextScaling appears to be a simple flow, consisting of setter *setTextScaleX()* and getter *getTextScale()*. However, the setter stores the transferred value into a variable managed by native code, which disables the TaintDroid's taint propagation at JNI-method-call edges. A goal of this chapter is to make it difficult for attackers to evade the taint tracking by disabling such uncomplicated techniques not specific to a particular topic. Also, similar techniques were discovered in some real-world apps [74]; thus, the current approaches discussed in Section 3.1.1 are unsafe.

In order to reliably detect information flows against the ATA techniques, a taint tracker should reduce the dependency on manually created native-method summaries. As yet, CTT [76] is the only currently available solution against the ATA techniques, specifically side channels, in ScrubDroid. This approach employs a hand-picked list of flow-causing API methods that only cover the ATA techniques in ScrubDroid. The list is incomplete because the information might flow through any API method provided by Android. Since most API methods are executed for purposes other than transferring information, the CTT's approach must produce FPs. For example, CTT propagates taint from the argument register of *setTextScaleX()* to the return value of *getTextScaleX()* of the TextView class. To suppress FNs, once a tainted value is passed to the setter, CTT propagates the taint to the return values of all subsequent getters. However, information flow from the setter occurs only when the getter's TextView instance has the same resource ID as the setter's; otherwise, no information flow occurs, and overtainting is caused. In addition, the list depends on the versions of the Android platform, and security analysts should devise a list of flow-causing API methods depending on the versions of their target apps and environments.

3.2 Four Flow Types

The characteristics of ATA techniques' flows (i.e., how a value for an untainted register is derived from the value of a tainted register) is leveraged to detect them with reducing the dependency on the list of flow-causing API methods. Since the classification in ScrubDroid does not effectively reveal features of flows caused by the techniques, this section characterizes them by newly defined types of information flow based on channels discussed by Cavallaro et al. [71]. Information can flow with assignment instructions and other channel types: control flows, pointer indirection, and timing channels. The DEX bytecode provides branches (e.g., *if*) and *array-op* instructions so that an app can transfer information with control flows and pointer indirection on the Android platform. It also offers the *array-length* instruction that returns the size of an array and can be classified to none of the four channel types. Since pointer-indirection-based techniques exploit arrays, it is extended to a new class consisting of all array-based techniques, including the *array-length*. Timing-based flows are also workable on the Android platform, as ScrubDroid includes one of them. Hence, four types of information flow are considered: data-flow-based, referred to as direct-assignment (DA) flows; array-based, memory-operation (MO) flows; control-flow-based, control-dependence (CD) flows; and timing-based, timing-channel (TC) flows.

TABLE 3.1: Characterization of the techniques in ScrubDroid based on newly defined types. They are also classified according to whether it is visible ($\circ$) or invisible ($\bullet$). The marks $+$ and dt denote flows created with altering the original techniques and flows having data-transformation capability respectively.

Class	ScrubDroid Technique	Defined Type DA	MO	CD	TC
Control Dependence	Simple Encoding	-	-	$\circ_{dt}$	$\circ^{+}_{dt}$
	Count-to-X	-	-	$\circ_{dt}$	$\circ^{+}_{dt}$
	Exception-Error	-	-	$\circ_{dt}$	$\circ^{+}_{dt}$
Subversion of Benign Code	Shell Command	$\bullet_{dt}$	-	$\bullet^{+}_{dt}$	$\bullet^{+}_{dt}$
	File-Shell Hybrid	$\bullet_{dt}$	-	$\bullet^{+}_{dt}$	$\bullet^{+}_{dt}$
Side Channels	Timekeeper	-	-	$\bullet^{+}_{dt}$	$\bullet_{dt}$
	File Length	-	$\bullet^{+}_{dt}$	$\circ_{dt}$	-
	Clipboard Length	-	$\bullet^{+}_{dt}$	$\circ_{dt}$	-
	Bitmap Pixel	-	$\bullet_{dt}$	$\circ^{+}_{dt}$	-
	Direct Buffer	-	$\bullet_{dt}$	$\circ^{+}_{dt}$	-
	Text Scaling	$\bullet$	-	$\bullet^{+}_{dt}$	-
	Bitmap Cache	$\bullet$	-	$\bullet^{+}_{dt}$	-
Unclassified	File Last Modified	$\bullet$	-	$\bullet^{+}_{dt}$	-
	Lookup Table	-	$\circ_{dt}$	$\circ^{+}_{dt}$	-
	Remote Control	-	-	$\circ_{dt}$	$\circ^{+}_{dt}$
	Remote Dex	-	-	$\circ_{dt}$	$\circ^{+}_{dt}$
#Techniques	16	5	5	16	8

Table 3.1 shows the characterization of the techniques in ScrubDroid, including four techniques (from the File Last Modified to Remote Dex) that are unclassified in ScrubDroid. Each of them causes a flow of one of the four flow types. They are also tested for producing different types of flow from the originals and identified that they all can generate multiple types of flow. Empty cells are for infeasible pairs of a technique and a flow type. The flows are further discussed from two aspects that can affect detection difficulty: visibility and data-transformation capability. A visible flow indicates the dependency between registers on the runtime layer, and an invisible flow does not. Any data transformation are considered, including encoding, obfuscation, hashing, and encryption. A data-transforming flow can cause $x \Rightarrow y$ where x and y hold distinct values. Other flows are called non-data-transforming flows. Stinson et al. [93] devised a tracking method based on data contents to detect flows occurring through an unmonitored area. It has also been discussed that ATA techniques can evade the data-content tracking by transforming data [71]. Data-transformation capability in ATA techniques in the Android platform is investigated to clarify the effectiveness of a tracking technique utilizing data contents. This section explains the characterization of the ATA techniques.

Table 3.1 does not cover all techniques belonging to the three classes, and more techniques can be found. While the multiple flow types for each technique are considered, covering all techniques belonging to each of the three classes is out of the scope because such coverage requires extensive effort. In particular, side-channel techniques that do not transform the transferred data in ScrubDroid could be modified to enable the data-transformation

```
TextView tv = findViewById(R.id.a);
tv.setTextScaleX(x)
y = tv.getTextScaleX();
```

FIGURE 3.1: DA flow $x \Rightarrow y$ w/ the TextView class, which extends the View class.

```
String cmd = "echo "+x;
Runtime runtime = Runtime.getRuntime();
BufferedReader br = new BufferedReader(new InputStreamReader(
    runtime.exec(cmd).getInputStream()));
y = br.readLine();
```

FIGURE 3.2: DA flow $x \Rightarrow y$ w/ *echo* command.

capability. This paper follows ScrubDroid and does not necessarily cover all techniques belonging to the three classes.

3.2.1 DA Flows

This type of flow $x \Rightarrow y$ is caused by executing two instructions. The first instruction has x as an operand and passes its value to API calls. The value is eventually stored in memory. The second instruction loads the value from the memory into y. The dependency between the two instructions is called read-after-write dependency, in which the second instruction has the value stored by the first instruction as an operand. In the situation that the dependency exists across API calls, the flow is invisible to bytecode-level trackers, failing to track the flow.

Five techniques in Table 3.1 cause the dependency across API calls. They can be divided into two groups based on data-transformation capability: unmonitored memory, which cannot transform data, and unmonitored code, which can transform data.

Non-data-transforming flows

Three of them exploit unmonitored memory. The Text Scaling (Figure 3.1) and Bitmap Cache leverage the View class, which is the fundamental element for user interface components and mainly operated in API calls. Similarly, the File Last Modified abuses a file's metadata stored outside an app's runtime instance. Their flows are considered as non-data-transforming. In the techniques, a value remains the same after moving from a tainted register to an unmonitored memory until it is moved to another register. In other words, an app cannot append values to an unmonitored memory and can only overwrite the current value with a new value. Hence, values should be transferred from x to y one by one in the same order via an unmonitored memory, which is referred to as a one-value feature.

Data-transforming flows

In contrast, the Shell Command (Figure 3.2) and File-Shell Hybrid leverage system command, for instance, *echo* and *cat*, provided by the Linux system. These two commands are not data-transforming, but there are data-transforming commands, such as *base64* and *md5*. Therefore, all shell command executions must be considered data-transforming for a tracker

```
char table[] = new char[256];
table[x] = 'a';  // aput-op DEX bytecode
ByteBuffer byteBuffer =
  ByteBuffer.allocateDirect(256).order();
byteBuffer.put(x,'a'); // put(index,data)
for (int i = 0; i < 256; i++) {
  if(table[i] == 'a'){y1 = i;}
  if(byteBuffer.get(i) == 'a'){y2 = i;}}
```

FIGURE 3.3: MO flows $x \Rightarrow y1$ w/ array and $x \Rightarrow y2$ w/ class.

not knowing shell commands. An app can pass data to and obtain results from system commands through the standard input/output and files. Note that the contents of files are not regarded as unmonitored because TaintDroid tracks information flows through files. It was found that unmonitored code can also store or obtain information to or from unmonitored memory, for example, filenames. An example is shown in Section 3.2.4.

3.2.2 MO Flows

MO flows $x \Rightarrow y$ are caused by arrays and class instances, and the x's value is translated into memory addresses.

Array-based MO Flows

The DEX bytecode does not support pointer operations, but arrays are available for visible MO flows. The DEX bytecode provides a series of *array-op* instructions, such as *aget-op*, which returns an element of an array; *aput-op*, which stores a value to an array; and *array-length*, which returns the size of an array. Since they clearly indicate the relationship between addresses and cells, the flows are visible on the runtime layer.

The Lookup Table listed in Table 3.1 uses the value of x as an index to obtain data from an array into y. Note that TaintDroid tracks this flow with a propagation rule for the *aget-op* because it commonly occurs with translation tables for character conversion. However, it was found that the *aget-op* can be replaced with the *aput-op* without changing the program's semantics, and TaintDroid will fail to track it (flow $x \Rightarrow y1$ in Figure 3.3). The value a is copied to the position *table[x]*, and later the program checks which index of *table* contains a to get x back. The result is copied to $y1$. In addition, researches mentioning taint propagation for the aput-op do not suggest a rule that can track the flow. Section 3.7.2 discusses this. Furthermore, the position of a can be shifted by partially copying *table* to another array, and a value for $y1$ will be changed from the value of x. As such, this type of flow can transform data. Thus, a tracking technique that utilizes data contents is ineffective. This type of flow can be caused by not only arrays but also classes, which will be explained in Section 3.2.2.

Class-based MO Flows

The Android APIs provide classes that cause invisible MO flows. It was identified that the Bitmap Pixel and Direct Buffer, shown in Figure 3.3, are such classes (Table 3.1) based on the number of API method arguments. To cause MO flows, an API method should take at least two arguments: a value specifying a location and another value being stored (e.g., value x

```
 1 String x = getSecret();  // taint source
 2
 3 // Flow to detect
 4 String string = TextUtils.join(
 5                 "", Collections.nCopies(
 6                 Integer.parseInt(x), "A"));
 7 y = string.length();
 8
 9 // FP to avoid
10 if (x.length() == 0) {
11     throw new RuntimeException("empty");}
```

FIGURE 3.4: MO flow $x \Rightarrow y$ occurring via the string's length, and CD flow $x \Rightarrow$ exception, which a system should avoid tracking.

and a respectively at line 5 in Figure 3.3). When the value of x is used as a location, it is not stored in memory, which is the difference between MO and DA flows.

Such classes' instances are accessed via API methods, obscuring the relationship between addresses and cells and making the flows invisible. Tracking MO flows such as $x \Rightarrow y2$ in Figure 3.3 requires taint propagations from arguments to the base object and the base object to arguments at different invocations. They can obfuscate locations as well as arrays, for example, the ByteBuffer class provides *position()* and *slice()* methods for shifting the cell locations. Therefore, a data-content-utilized tracker is ineffective against class-based MO flows. Note that this characterization is given only in the app layer. In contrast, API methods may cause other types of flow, such as DA or CD, in their implementation. Since the approach analyzes only app code, this paper classifies information flows based on their characteristics in the app layer.

Length-based MO Flows

The value of x can be used to determine the size of a new array and is translated into the distance between two addresses. Then, a value for y is derived by measuring the array's length with the *array-length*. Class instances also can be used as demonstrated by the File Length and Clipboard Length in Table 3.1. ScrubDroid's original code encodes data with loops, whereas it can be altered by replacing the loops with the API methods (Figure 3.4). Length-based MO flows can transform data, for example, the values of *string* and *string.length()* is different in Figure 3.4. Hence, a data-content-utilized tracker is ineffective against length-based MO flows. A challenge is that since the length is commonly operated with branches, simply tracking all lengths [76] can open ample opportunities for false detection of CD flows. For example, apps check whether a string is empty (Line 10 in Figure 3.4). If a tracker propagates x's taint to the length and detects the CD flow, the taint is assigned to the exception. Subsequently, an FP detection occurs if the exception information is passed to a sink.

3.2.3 CD Flows

CD flows $x \Rightarrow y$ leverage control flows, determining what sequences of instructions can be executed. A branch is executed with x as an operand, and control is conditionally transferred depending on the value. A value for y is determined by control-dependent instructions

```
TextView tv1 = findViewById(R.id.a);
TextView tv2 = findViewById(R.id.a);
if (x == 1) { tv1.setTextScaleX(1); }
else { tv2.setTextScaleX(0); }
y = tv1.getTextScaleX();
```

FIGURE 3.5: CD flow $x \Rightarrow y$ occurring through unmonitored memory that is used in the Text Scaling technique.

executed between the branch and its re-convergence point (i.e., immediate post-dominator of the branch).

Visible CD Flows

If y is the destination operand of a control-dependent assignment instruction, the dependency is visible in the app code. It is implemented in the Simple Encoding, Count-to-X, Exception-Error, Remote Control, and Remote Dex (Table 3.1). Techniques causing MO flows in Table 3.1 can also cause visible CD flows in which the destinations are the same register on both sides of a branch.

Invisible CD Flows

It was discovered that the rest of the techniques in Table 3.1 also can be used as a part of CD flows. The five techniques of DA flows can generate invisible CD flows that a control-dependent instruction in one side of a branch stores an untainted value into unmonitored memory, and a value for y is derived by an instruction executed after the re-convergence point. Another control-dependent instruction on the other side of the branch can store another untainted value into the same unmonitored memory with a separate instance, for example, two instances of the TextView class *tv1* and *tv2* are used depending on whether a branch is taken, and information is invisibly transferred from x to y (Figure 3.5).

Information Preservability

For precise tracking, it is important to track only CD flows $x \Rightarrow y$ where x is a branch's operand, and y is assigned a value in one side of the branch and another distinct value in the other side (i.e., a value for y is conditionally determined depending on the value of x) [91]. Such flows can transfer a certain amount of information and are called information-preserving CD flows. Therefore, a challenge in this paper is to distinguish between information-preserving and non-information-preserving CD flows. You et al. [64] developed a technique that inspects the information preservability of visible CD flows by checking values of registers at re-convergence points. Suppose a register has a unique value depending on whether or not a branch is taken. In that case, the register is detected as the destination of an information-preserving CD flow and is subsequently tainted. However, their approach will fail in the case that API calls are exploited to cause control-dependent DA flows (i.e., invisible CD flows). Information can flow not to registers but across API calls, which cannot be examined at the re-convergence points. For example, *tv1.setTextScaleX(1)* is executed if the condition is true in Figure 3.5; otherwise, *tv2.setTextScaleX(0)* is executed. In order to perform the information-preservability inspection, a system must know whether the destinations (i.e., the stored memory locations) of the two methods' arguments are the same.

```
File file = new File("/secret/"+x);
file.createNewFile();
String cmd = "sleep $(ls /secret)";
long start = System.currentTimeMillis();
Runtime.getRuntime().exec(cmd).waitFor();
y = System.currentTimeMillis() - start;
```

FIGURE 3.6: TC flow $x \Rightarrow y$ with the *sleep* command that takes an argument from the filename, which equals to x's value.

However, a system cannot simply decide it based on the method names because it depends on *R.id.a*. In other words, the challenge is to equate the destinations of the two control-dependent invocations or to detect the flow without the equating.

3.2.4 TC Flows

This type of flow $x \Rightarrow y$ encodes data into timing by delaying program execution for a while, depending on the x's value. Then, the delay time is measured and saved into y. TC flows cause no data flow but can transfer information. Since the current solution, CTT, tracks TC flows, TC flows are considered in this paper.

Visible TC Flows

Delays can be produced with loops and be measured by communication intervals on a remote server [71]. Hence, TC flows are implemented with loops based on the Simple Encoding, Count-to-X, Exception-Error, and Remote Dex. Also, the Remote Control is altered into a technique measuring delay times from a remote server. They all cause visible flows because loops consist of branches, which are identifiable in the app code.

Invisible TC Flows

The Timekeeper in Table 3.1 invokes *sleep()* API method to produce delays. The API method can be replaced by any API method that performs similarly to the *sleep()*. Thus, their flows are invisible. In addition, it was found that API calls supply more options for apps to cause TC flows. The Linux system provides a shell with that *sleep* and *for* commands can be executed apart from app code, and invisible flows occur (Figure 3.6). The technique is implemented based on the Shell Command and File-Shell Hybrid in Table 3.1, whose execution time can be changed depending on its arguments. A system that only focuses on specific API methods on the runtime layer, such as [76], will fail to detect invisible TC flows. Even if a system considered *Runtime.getRuntime().exec(cmd).waitFor();* a TC-flow-causing API, the taint propagation could not be performed because argument *cmd* is not tainted. A challenge is to detect the flow and propagate the taint of x.

3.3 Approach

The approach tracks visible and invisible flows of DA, MO, CD, and TC to detect all the techniques in ScrubDroid and other techniques exploiting unknown media across API calls. The tracking reduces the dependency on the API method list and versions of the Android

Algorithm 1 Taint propagation across an API call

```
1:  procedure PROPAGATE(method)
2:      $bo \leftarrow$ method's base object
3:      $args \leftarrow$ invoke-op's arguments including bo
4:      $ret \leftarrow$ move-result-op's destination register
5:      if method's name ends with "length" or "size" then
6:          if bo is tainted and
7:             bo's value differs from taint source values then
8:              assign bo's taint to ret
9:          else
10:             remove ret's taint
11:         end if
12:         return
13:     end if
14:     for arg in args do
15:         if arg is tainted then
16:             perform the value logging with arg
17:             perform the TC encoder detection with arg
18:         end if
19:     end for
20:     remove ret's taint
21:     $copied\_args \leftarrow$ copy of args before taint propagation
22:     for copied_arg in copied_args do
23:         if copied_arg is tainted then
24:             for arg in args do
25:                 assign copied_arg's taint to arg
26:             end for
27:             assign copied_arg's taint to ret
28:         end if
29:     end for
30:     if ret is not tainted then
31:         perform the value matching with ret
32:         if method is System.currentTimeMillis() then
33:             perform TC decoder detection with ret
34:         end if
35:     end if
36: end procedure
```

platform. Hence, it covers more flow-causing API methods than CTT. The tracking also reduces the number of FPs by inspecting whether the information is transferred. Furthermore, binary-level tracking is excluded to leverage the variable semantics.

Algorithm 1 shows how the approach propagates taint across an API call. First, it performs the length tracking (Section 3.3.2) in Lines 5-13. If the target method is detected as a length-related method, the procedure ends with the *return* at Line 12. Next, the value logging (Section 3.3.1) and TC encoder detection (Section 3.3.4) are performed in Lines 14-19. Then, the class tracking (Section 3.3.2) is executed in Lines 20-29, and the value matching (Section 3.3.1) and TC decoder detection are performed (Section 3.3.4) in Lines 30-35. Additionally, the approach operates the array tracking (Section 3.3.2) and the CD-flow tracking (Section 3.3.3) to detect information flows across array operations and control dependencies.

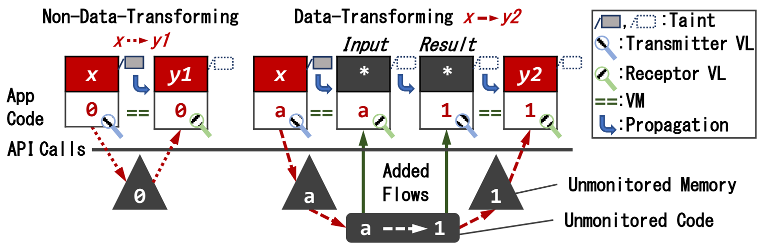


FIGURE 3.7: Two DA flows: $x \Rightarrow y1$ and $x \Rightarrow y2$. The value logging (VL) and matching (VM) detect flows and propagate the taints between x and $y1$, x and *Input*, and *Result* and $y2$. In contrast, *Input*'s taint is transparently propagated to *Result*.

3.3.1 DA-Flow Tracking

The DA-flow tracking exploits actual data values transferred across API calls to track invisible flows. It logs operand values of any API method invocation to not rely on the list of flow-causing API methods. Also, it matches the values of tainted and untainted operands and propagates the taint between them.

Value Logging

In order to capture all values passed to and obtained from API calls, the DA-flow tracking intercepts operand values of each *invoke-op (API)* instruction that calls an API method. Note that the *invoke-op* also calls methods implemented within app code, referred to as *invoke-op (non-API)*.

For non-data-transforming flows The value logging is performed before each execution of the *invoke-op (API)* as follows:

1. When the *invoke-op (API)* is about to be executed, the tracker checks taint of its operand registers, denoted x in Figure 3.7.
2. It logs values of tainted registers among them and flags them as candidates for a transmitter passing values to API calls.
3. Logged values are kept separately by instruction addresses and register names.

Simultaneously, the value logging is performed after each execution of the *invoke-op (API)* as follows:

1. When the *invoke-op (API)* is terminated, the tracker checks taint of values produced by the method that are its return value and values of reference-data-type arguments, to which the method might append values, denoted $y1$ and $y2$ in Figure 3.7.
2. It logs untainted values among them and labels them as candidates for a receptor that obtains values from API calls.
3. Logged values are kept in the same way as transmitter candidates.

For data-transforming flows The logging before and after the *invoke-op (API)* is sufficient to detect $x \Rightarrow y1$ in Figure 3.7 with the value matching. However, the logging is ineffective when the data is transformed , for example, $x \Rightarrow y2$ in Figure 3.7, where their values are different. The unmonitored code obtains a value *a* from the unmonitored memory, encodes it to *1*, and stores it to the unmonitored memory. The encoding is done by *grep -c a*, and more data transformation can be performed by other shell commands (e.g., *base64* and *md5*).

The tracker, therefore, logs more values at each invocation of the API method executing shell commands (e.g., *Runtime.exec()*). It injects bytecode that creates additional flows from the shell commands to the app layer for capturing values accessed and outputted by the shell commands. Then, it processes the values depending on the taint of the invoked API method's argument. If the argument is not tainted, it considers each value as a receptor candidate (i.e., input of unmonitored code). For example, assume that the unmonitored code in Figure 3.7 is executed by an API method invocation with an untainted argument, it logs the value *a* as the input of the unmonitored code (denoted *Input*) and flags the input as a candidate for a receptor.

If the argument is tainted or its input is tainted, the tracker considers each value as a transmitter candidate (i.e., the result of unmonitored code). For example, assume that the input of the unmonitored code in Figure 3.7 is tainted, it logs the value *1* as the result of the unmonitored code (denoted *Result*) and flags *Result* as a candidate for a transmitter.

Value Matching

This operation detects DA flows by comparing captured values. When a new value is logged as a receptor candidate, the tracker checks whether the value equals the last logged value of each transmitter candidate. If two values of a transmitter and receptor candidates are identical, the taint is propagated between them. For example, a candidate pair *x* and *y1* hold the same value *0* and are matched in Figure 3.7. Values must match between a transmitter and a receptor; however, there is an exceptional technique called Bitmap Cache, which transfers a value by showing it on the screen and capturing it into a screenshot. The image is logged at a receptor. The tracker detects an image and extracts the value printed on the image by using optical character recognition. The tracker uses the value as the receptor's value.

In contrast, another pair *x* and *y2* hold different values *a* and *1* because the unmonitored code transforms the data. The tracker tracks the flow as follows:

1. It flags *x* as a transmitter candidate and *Input* as a receptor candidate as described in Section 3.3.1.

2. Subsequently, it compares values of *x* and *Input* and propagates taint between them because their values, *a* and *a*, are identical.

3. Since *Input* is tainted, the other logged value *1* of the unmonitored code is flagged as a candidate for a transmitter (*Result*), and the *Input*'s taint is propagated to *Result*.

4. When the value of *y2* is logged, the tracker compares *y2*'s value with the last logged value of each transmitter candidate, including *Result*, and the *Result*'s taint is propagated to *y2* because their values, *1* and *1*, are identical.

TABLE 3.2: Taint-propagation rules for MO flows caused by the *array-op* and the *invoke-op (API)*. Function $\tau(v)$ returns taint of register variable v or assigns taint to it. V_{Re} and V_{Pr} are a set of reference-data-type and primitive-data-type arguments respectively, and $\tau(V)$ returns the union of all registers' taint in V or assigns taint to all of them. Rules marked $*$ are conditionally applied depending on the operand's value.

#	Instruction	Taint Propagation
1	*aput-op* $v_{Src}, v_{Array}, v_{Idx}$	$\tau(v_A) \leftarrow \tau(v_S) \cup \tau(v_A)$
2		$\tau(v_A) \leftarrow \tau(v_I) \cup \tau(v_A)$
3	*aget-op* $v_{Dst}, v_{Array}, v_{Idx}$	$\tau(v_D) \leftarrow \tau(v_I) \cup \tau(v_A)$
4		$\tau(v_I) \leftarrow \tau(v_I) \cup \tau(v_A)$
5	*array-length* v_{Dst}, v_{Array}	$\tau(v_D) \leftarrow \tau(v_A)^*$
6	*invoke-op (API)* $\{V_{Re}, V_{Pr}\}$	$\tau(V_{Re}) \leftarrow \tau(V_{Re}) \cup \tau(V_{Pr})$
7		$\tau(V_{Pr}) \leftarrow \tau(V_{Re}) \cup \tau(V_{Pr})$
8		$\tau(result) \leftarrow \tau(V_{Re}) \cup \tau(V_{Pr})$
9	*move-result-op* v_{Dst}	$\tau(v_D) \leftarrow \tau(result)^*$

3.3.2 MO-Flow Tracking

Since MO flows can transform data as described in Section 3.2.2, the value logging and matching are ineffective. Therefore, the rules in Table 3.2 are used to propagate taint for arrays and class instances.

Array Tracking

Visible MO flows transfer information between components and indices of arrays. Since *array-op* instructions of the DEX bytecode indicate value, array, and index registers, the relationship among them is easily detected, and taint propagation for them is achieved straightforwardly. When the *aput-op* is executed, the source operand's value is moved into the array depending on the index operand's value, and no register is completely overwritten. Thus, any register's taint must not be removed. As shown in Table 3.2, the array tracker appends taint of the source and index registers to the array register for the *aput-op* (Rules 1 and 2). When *aget-op* is executed, the destination operand's value is completely overwritten with a value depending on the array and index operands. Hence, the tracker overwrites the taint of the destination register with the taint of the array and index registers (Rule 3). At the same time, taint of the array register is appended to the index register by Rule 4, which enables a tracker to follow the flow $x \Rightarrow y1$ in Figure 3.3.

Rule 4 is unnecessary if a tracker performs CD-flow tracking at the same time because there is a CD flow from the branch's operand *table[i]* to *i* at line 7 in Figure 3.3. However, both Rule 4 and CD-flow tracking are employed altogether to monitor the number of occurrences of each flow and identify the path of information transfer more clearly for further investigation.

Class Tracking

The approach tracks invisible MO flows by conservatively propagating taint among operand registers of the *invoke-op (API)*. It covers all the MO-flow techniques in Table 3.1 and unlisted techniques exploiting unknown API classes. An app-level taint tracker [65] utilizes taint propagation rules for API methods, only considering reference-data-type arguments to minimize the impact of overtainting. The restriction is loosened to support primitive-data-type arguments and track MO flows more comprehensively. As shown in Lines 20-29 in Algorithm 1, at each *invoke-op (API)* execution, the taint is propagated from any operands to all reference-data-type arguments and the return value of the invoked method, which is the destination register of the *move-result-op* instruction (Rules 6, 8, and 9 in Table 3.2). Since operands of the *invoke-op (API)* invoking a non-static method contain an instance of the invoked method's class (i.e., a base object), the rules track such instances as well as arguments. The rules are based on over-approximations and could generate FPs. The rules will be evaluated to show that the rules generate fewer FPs than CTT in Section 3.5.4.

Rule 7 propagates the taint from reference-data-type to primitive-data-type arguments (Table 3.2) to track MO flows such as $x \Rightarrow y2$ occurring via instance *byteBuffer* and primitive-data-type argument i (Figure 3.3). Rule 7 is unnecessary if CD-flow tracking is simultaneously performed, but both Rule 7 and CD-flow tracking are implemented for the same reason as Rule 4. Rules 6-9 also track MO flows when the instance *byteBuffer* is converted into an array by an API method after line 5 instead of causing a CD flow in Figure 3.3.

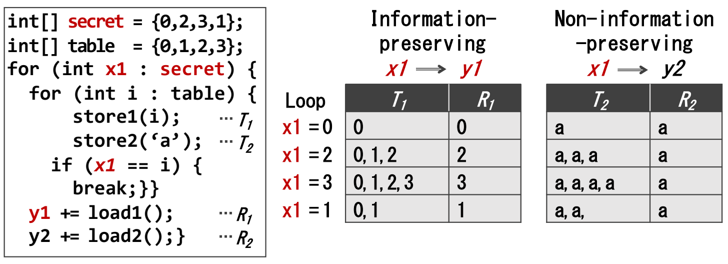
Length Tracking

To reduce overtainting for the *array-length*, the length tracker utilizes data values to propagate the taint from the array to its length (Rule 5 in Table 3.2). The tracker propagates the taint only if a value joining all elements of an array is not the same as the original values of tracked information (i.e., values loaded into registers at taint sources). This rule is based on the fact that a value needs to be tracked only if its length presents the tracked information, and the length of the original value itself will never present the information. If the propagation is skipped, the tracker simply removes the taint of the destination operand.

Similarly, when the *invoke-op* calls a length-related method, it propagates taint from a class instance (i.e., an operand of the *invoke-op (API)*) to its length (i.e., the destination register of the corresponding *move-result-op*) only if the value of the instance is not identical to original values of the tracked information (Rule 9 in Table 3.2). For example, the variable *string*'s value does not match with the variable x's value at the taint source in Figure 3.4, and the tracker propagates the *string*'s taint to y at Line 7. On the other hand, x at Line 10 contains the same value as one at the taint source, the tracker does not propagate the x's taint to the length, and the CD-flow tracking is not performed for the *if* statement at Line 10. Length-related methods are detected by searching words *length* and *size* in their name (Line 5 in Algorithm 1). Note that the search relies on API method names not to reduce FNs but to reduce FPs. If the tracker fails to detect a length-related method (i.e., the condition is false in Line 5 in Algorithm 1), the conservative rules will taint the length in Lines 20-29 in Algorithm 1. The propagation causes a true positive (TP) if the length preserves the target information. Otherwise, it generates an FP. Hence, FNs do not occur.

3.3.3 CD-Flow Tracking

The approach tracks CD flows on the runtime layer and across API calls taking the information preservability into account as discussed in Section 3.2.3. The CD-flow tracker first identifies control dependencies at the bytecode level to detect control-dependent instructions. Then, the tracker inspects whether control-dependent instructions preserve the information of the corresponding branch's operand. It propagates the taint of the branch's operand to destinations of the control-dependent instructions.



```
int[] secret = {0,2,3,1};
int[] table  = {0,1,2,3};
for (int x1 : secret) {
  for (int i : table) {
      store1(i);     ··· T1
      store2('a');   ··· T2
    if (x1 == i) {
      break;}}
  y1 += load1();     ··· R1
  y2 += load2();}    ··· R2
```

	Information-preserving $x1 \longrightarrow y1$		Non-information-preserving $x1 \longrightarrow y2$	
Loop	T_1	R_1	T_2	R_2
x1 = 0	0	0	a	a
x1 = 2	0, 1, 2	2	a, a, a	a
x1 = 3	0, 1, 2, 3	3	a, a, a, a	a
x1 = 1	0, 1	1	a, a,	a

FIGURE 3.8: Two invisible CD flows of the first type, $x1 \Rightarrow y1$ and $x1 \Rightarrow y2$, which are direct-control-dependence-based and caused by the *if* statement. T_n and R_n are transmitters and receptors of the DA flows respectively (n = 1, 2). API method *loadN()* returns values transferred by *storeN()* (N = 1, 2). Tables list the transitions of the logged values in each iteration of the outer loop.

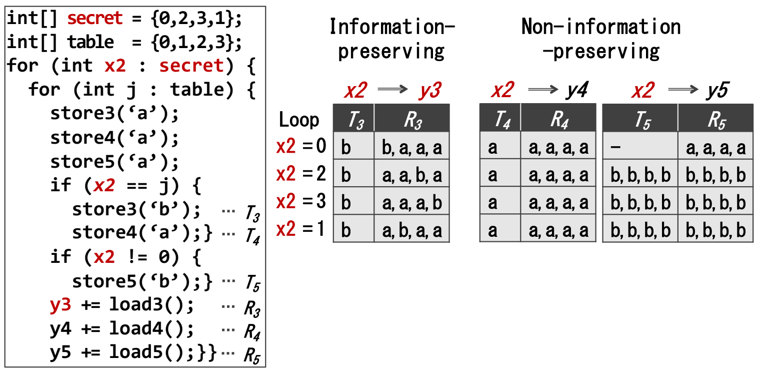
Regardless of the visibility, CD flows are divided into two types: direct-control-dependence-based and implicit flows [71]. To illustrate examples, Figure 3.8 shows two invisible CD flows categorized as direct-control-dependence-based, and Figure 3.9 gives three invisible CD flows categorized as implicit flows. In Figure 3.8, transmitters T_1 and T_2 are directly control-dependent on the *if* statement because the transmitters are in the inner *for* loop's body, and the loop is exited by the *if* statement. In contrast, some of the *storeN()* invocations (i.e., transmitters of the DA flows) in Figure 3.9 are indirectly control-dependent on the *if* statements because they are outside of the *if* statements' blocks, and the *if* statements have no control over the *for* loops, but the corresponding receptors can receive values depending on the *if* statements. Also, visible CD flows are divided into either direct-control-dependence-based or implicit according to whether the dependency is direct or indirect.

Visible-CD-Flow Tracking

For visible CD flows, the tracker checks whether the information is preserved between a branch's operand and a register assigned values by control-dependent instructions as follows:

Assigned register inspection for direct-control-dependence-based flows The tracker checks whether a register is assigned a value on one side of the branch depending on the value of the branch's operand. The examination is based on the notion of distinct code [94]. The branch's operand values are considered words, and the values assigned to the register are regarded as codewords. If the mapping from words to codewords is one-to-one, the flow must preserve information because all of the words can be correctly derived from the codewords. If the flow is information-preserving, the tracker propagates the taint from the branch's operand to the register at the re-convergence point.

Branch's untainted operand inspection for implicit flows If the taint is not propagated in the previous inspection, the tracker checks whether a register is assigned a different value depending on whether the branch is taken or not taken. If so, the tracker inspects whether the values of the branch's untainted operand differ depending on the values of the branch's tainted operand when the branch is taken. If the flow satisfies these requirements, the tracker considers the flow as information-preserving and propagates the taint from the branch's operand to the register at the re-convergence point.



```
int[] secret = {0,2,3,1};
int[] table  = {0,1,2,3};
for (int x2 : secret) {
  for (int j : table) {
    store3('a');
    store4('a');
    store5('a');
    if (x2 == j) {
      store3('b');   ··· T3
      store4('a');} ··· T4
    if (x2 != 0) {
      store5('b');} ··· T5
    y3 += load3();   ··· R3
    y4 += load4();   ··· R4
    y5 += load5();}} ··· R5
```

	Information-preserving		Non-information-preserving			
	$x2 \longrightarrow y3$		$x2 \longrightarrow y4$		$x2 \longrightarrow y5$	
Loop	T_3	R_3	T_4	R_4	T_5	R_5
x2 = 0	b	b, a, a, a	a	a, a, a, a	–	a, a, a, a
x2 = 2	b	a, a, b, a	a	a, a, a, a	b, b, b, b	b, b, b, b
x2 = 3	b	a, a, a, b	a	a, a, a, a	b, b, b, b	b, b, b, b
x2 = 1	b	a, b, a, a	a	a, a, a, a	b, b, b, b	b, b, b, b

FIGURE 3.9: Three invisible CD flows, *x2* ⇒ *y3*, *x2* ⇒ *y4*, and *x2* ⇒ *y5*, which are implicit flows and resulting from the *if* statements. T_n and R_n are transmitters and receptors of the DA flows respectively (n = 3, 4, 5). API method *loadN()* returns values transferred by *storeN()* (N = 3, 4, 5). Tables list the transitions of the logged values in each iteration of the outer loop.

Invisible-CD-Flow Tracking

In order to track invisible CD flows, control-dependent DA flows should be detected, as Section 3.2.3 explained. When an API method is invoked on a side of a branch, and the branch's operand is tainted, the invisible-CD-flow tracker makes the method's arguments transmitter candidates of DA flow. The value logging and matching detect control-dependent DA flows between the transmitters and receptors executed after the re-convergence point.

When a control-dependent DA flow is detected, the tracker examines the information preservability of the invisible CD flow. Since control-dependent instructions in the CD flow assign values not to registers but to API calls, the information preservability cannot be examined in the same way as the visible-CD-flow tracker.

Figure 3.8 and Figure 3.9 show invisible CD flows that are direct-control-dependence-based and implicit flows, respectively. In each example, Transmitters T_n are executed within the inner loop's body, but receptors R_n are invoked outside the inner loop's body. In this case, the only values transferred by the transmitters in the last iteration of the inner loop are captured by the receptors because of the one-value feature, which an app can only overwrite an unmonitored memory as Section 3.2.1 explained.

Figure 3.8 shows two invisible CD flows, which are direct-control-dependence-based. In flow *x1* ⇒ *y1*, transmitter T_1 transfers values, *0, 2, 3,* and *1*, depending on tainted operand *x1*'s values, *0, 2, 3,* and *1*, and the information is preserved. In contrast, the information is not preserved if the transmitter's values do not depend on the tainted values (*x1* ⇒ *y2*). Figure 3.9 illustrates three invisible CD flows, which are implicit flows. In flow *x2* ⇒ *y3*, transmitter T_3 is executed with values depending on the comparison between tainted operand *x2*'s values, *0, 2, 3,* and *1*, and untainted values, *0, 1, 2,* and *3*, in *table*, of which the attacker has known the content. The attacker can restore the information from the sequence of the receptor's values, *a* and *b*. If the transmitter's values do not depend on the comparison results, such as T_4 of *x2* ⇒ *y4*, the flow does not preserve the information. Also, the information is not preserved by *x2* ⇒ *y5* because the branch's condition *x2 != 0* does not produce a one-to-one mapping between *x2*'s values and the T_5 execution.

Transmitter inspection for direct-control-dependence-based flows When a DA flow with control dependency is detected, the tracker checks whether the transmitter's values differ depending on the tainted values of the branch. The tracker utilizes not all values logged at the transmitter but only values that appeared at the receptor. If the dependency is detected, it propagates the taint of the branch's operand to the receptor of the DA flow. Flow $x1 \Rightarrow y1$ preserves the information because the transmitter's values in the inner loop's last iterations (i.e., the receptor's values), *0, 2, 3,* and *1*, differ depending on *x1*'s values, *0, 2, 3,* and *1*. Hence, the tracker propagates *x1*'s taint to R_1 and *y1*. In contrast, flow $x1 \Rightarrow y2$ does not preserve the information because receptor R_2 always receives a, regardless of *x1*'s values.

Branch's untainted operand inspection for implicit flows If the taint is not propagated in the previous inspection, the tracker checks whether the receptor's values contain any value that is not contained in the transmitter's values. If so, the receptor contains distinct values transferred from other transmitters, and the information can be preserved. With the checking, the tracker can proceed to the information-preservability inspection without the equating of the destinations of control-dependent instructions. The tracker inspects whether the values of the branch's untainted operand differ depending on the values of the branch's tainted operand when the branch is taken. If there is a one-to-one mapping between values of the untainted and tainted operands, the tracker considers the flow as information-preserving and propagates the taint from the tainted operand to the receptor. Flow $x2 \Rightarrow y3$ preserves the information because receptor R_3 obtains value a, which is different from values transferred from T_3. Additionally, values of the branch's untainted operand j change depending on values of the branch's tainted operand *x2* when the branch is taken. Hence, the tracker propagates the taint from *x2* to *y3*.

In contrast, consider flow $x2 \Rightarrow y5$, which the branch's condition is *x2 != 0* instead of *x2 == j*. The receptor obtains the same sequence of values (b, b, b, b) when *x2*'s value is *1, 2,* and *3*, and the information is not preserved. The tracker detects it based on the branch's operand values, which untainted value *0* does not change depending on *x2*'s values, and the taint is not propagated. Also, flow $x2 \Rightarrow y4$ is non-information-preserving because receptor R_4 only contains values a, which is the same value as T_4. Note that the CD-flow tracker does not detect so-called hidden implicit flows. Section 3.6.1 discusses this further.

3.3.4 TC-Flow Tracking

The tracker detects TC flows by detecting TC encoders and decoders and calculating the correlation between them. TC encoders are instructions delaying the app execution depending on tainted values, and the TC-flow tracker detects TC encoders without a fixed list of API methods (e.g., *Thread.sleep()*) and shell commands (e.g., *sleep*) as follows:

1. When the *invoke-op (API)* is executed with a tainted operand, the tracker calculates the time taken by the invocation. When the *invoke-op (API)* is executed without tainted operands, the value matching is used to detect shell commands loading a tainted value from not the argument but unmonitored memory. Then, the API invocation executing such shell commands is detected as a TC encoder. If the same encoder is consecutively executed with the same tainted value, the times are added up.

2. Then, the tracker computes the correlation between the tainted operand's values and the times. It uses a threshold to determine whether there is a one-to-one mapping between the tainted values and the times. If so, the tracker considers the API method invocation as a TC encoder.

3. For *if* instruction, which can generate loops and delay the app execution, the tracker simply flags it as a TC encoder when its operand is tainted.

The TC-flow tracker detects TC decoders and calculates the correlation between TC encoders and decoders as follows:

1. When *System.currentTimeMillis()* API method, which returns the system clock, is invoked, the tracker considers the invocation as a TC decoder. Also, it considers sinks as TC decoders.

2. When TC decoders are executed multiple times, the tracker first calculates the time interval between two TC decoders that are executed immediately before and after a TC encoder.

3. Then, it computes the correlation between the intervals and values passed to the TC encoder. Even if the exact value is passed to a TC encoder, there will be variations in their durations [95]. Therefore, a threshold is used to determine whether durations are distinguishable depending on the operand values of a TC encoder. If so, the tracker propagates the taint from the TC encoder's operand to the return value of the TC decoder. If the TC decoder is a sink, it produces an alert for privacy leak detection.

3.4 Implementation

A taint analysis system called VTDroid is developed with employing the tracking technique to detect privacy leaks and input validations in apps. VTDroid executes taint analysis on a server to make the implementation independent of specific versions of Android and devices. Since any sufficient dynamic-taint-tracking tool for real-world app analysis was found, the taint propagation rules were decided to be implemented on VTDroid. For example, TaintDroid is only available on Android version 4.3 or earlier and cannot be used to analyze apps requiring the newer Android versions. VTDroid also has the advantage that performing taint analysis on a server makes it possible for users to efficiently test taint propagation rules without running target apps every time. Also, VTDroid requires no device modification, such as rooting and makes analysis details accessible to its users.

3.4.1 Overview

VTDroid consists of an analysis server and an Android device. First, the server extracts smali files, written in an assembly language of the DEX bytecode, from an app and statically instruments code into them (Section 3.4.2). Then, the analyst re-creates the app with the modified smali files and a self-signed certificate. Next, the analyst installs and launches it on the device, and also the added code is executed and records the app's runtime data to a file on the device's storage. Finally, the server obtains the file to reproduce the app's behaviors and performs taint analysis (Section 3.4.3). It also statically analyzes control flows of the app's smali files (Section 3.4.4).

A difference between VTDroid and the other dynamic analysis tools, TaintDroid and CTT, is that only VTDroid records the app's execution trace. The execution trace provides information about executed code and processed values to analysts and makes understanding the app's behaviors easier. Since the taint tracker is not executed on the device, only the

TABLE 3.3: VTDroid injects code into positions before and/or after instructions and labels. Argument, index, source, and destination indicate target operand registers for recording values, and $*$ indicates points at which the code is injected to record the code execution without recording values.

Instruction or Label	**Target Register**	
	Before	**After**
invoke-op	argument	argument
move-result-op	-	destination
aput-op, *aget-op*	index	-
aput-op, *iput-op*, *sput-op*, *cmp-op*	source	-
aget-op, *iget-op*, *sget-op*, *unop*, *binop*	-	destination
if, *switch*	$*$	$*$
switch-L, method head	-	$*$

recording can affect the app runtime overhead. The app runtime overheads of the tools are compared in Section 3.5.6.

3.4.2 Static Bytecode Instrumentation

Before the app installation, the server instruments code that saves the app's runtime data into the device's storage, enabling the server to emulate the app's behaviors. Recorded information is as follows:

- timestamp of the bytecode execution
- process identifier (PID) of a process executing the bytecode
- thread identifier (TID) of a thread executing the bytecode
- identifiers of code-instrumented class and method
- smali file's line number at that the code is instrumented
- name and value of registers.

Table 3.3 lists DEX instructions, positions, and registers for that the code is instrumented with or without recording values. The value-based taint propagation tracks information transferred by the ATA techniques. Such information is, for example, in a format that can be passed to a shell command or processed by a conditional instruction. Currently, techniques that can transfer arbitrary objects are not found. Therefore, the code records the name and value of registers only in the case that its type is convertible to the String class, covering all eight primitive data types, String, and arrays of any of the nine types. In addition, values of the android.text.Editable type are logged at the invocation of taint sources for suspicious validation detection. The server statically extracts bytecode constants (i.e., *const-op*) from smali files as necessary, and unconditional jumps (i.e., *goto-op*, *return-op*, and *throw-op*) are also statically analyzed without the code instrumentation.

The server also checks whether methods are implemented within the app to classify API and non-API methods. Since the relationship between threads should be explicitly recorded, VTDroid employs a technique [64] creating an extra class field to methods that generate a new thread (e.g., *AsyncTask.execute()*). It instruments code at a caller's side that saves TID to the field. It also instruments code that loads the field's value at a callee's side and writes it to the record file at the called method's head.

As explained in Section 3.3.1, VTDroid instruments bytecode right before the *invoke-op (API)* calling *Runtime.exec()*, which executes shell commands. The bytecode obtains the executed shell commands from the invoked method's argument register and processes the commands as follows:

1. The commands are obtained as a string, and the bytecode parses the string from the head.
2. When a pipe is found, the bytecode extracts and executes the part from the head to the current position without the pipe, and the standard output is recorded.
3. When a "$(" is found, it extracts and saves the part from the head to the current position into a stack.
4. When a ")" is found, it extracts and executes the part from the previously found "$(" to the current position without the dollar and parentheses and records the standard output. Then, the bytecode concatenates the part with the element popped from the stack.
5. When the bytecode parses the string to the end, it executes the entire string and records the standard output.

In addition, after each execution of a part, the bytecode checks whether the part contains the slash. When the slash is found, the bytecode considers that the part contains a file path. It extracts the file path and records the file's content by executing the *cat* command with the path. As a result, VTDroid can capture the output of shell commands when the output is redirected to a file.

3.4.3 Taint Analysis

The app is exercised on the device, and its runtime data is delivered to the server. Based on recorded PIDs, TIDs, class and method identifiers, and line numbers, the server follows the traces from the dynamic execution, including caller-callee relations of non-API methods, taken paths at conditional branches, and exception occurrences. Then, it reproduces the sequence of instructions executed on the device and simultaneously performs data-flow tracking and ATA-technique detection.

Data-Flow Tracking

The server tracks data flows resulting from the DEX bytecode assignment instructions with the same propagation rules as those of TaintDroid. In addition, the taint is propagated from the source to destination registers for the *cmp-op* instruction. At the same time, the server calculates and sets the result of the *cmp-op* to the destination register by simulating the instruction.

When tainted registers are passed to a non-API method called by the *invoke-op (non-API)*, the server propagates taint between arguments and parameters of the invoked method, and its return value's taint is propagated to the destination register of the corresponding *move-result-op* executed by the caller. At the same time, taint is propagated from callee's parameter registers to the caller's reference-data-type arguments if the callee appends a tainted value to the parameter registers. For JNI methods, the server currently employs the same propagation rules as those of API methods.

The server exploits actual values for some propagation rules. Class instance fields and static fields are tracked based on their name and value. Similarly, it currently tracks data flows through files, the SharedPreferences, and inter-component communication based on their contents. Note that the tracking through files is limited. The system tracks data flows through files by the value logging and matching, which is effective as long as the one-value feature is satisfied. Since files should not always satisfy the one-value feature, the system can miss data flows through files.

ATA-Technique Detection

Simultaneously with the data-flow tracking, the server executes the tracking technique described in Section 3.3 at each execution of the *invoke-op (API)* and branches, which is explained in Section 3.4.4. When the *invoke-op (API)* is executed, the server operates the value logging and matching if the invoked method is not one of the taint sources, sinks, and length-related methods. The value logging currently targets the data types explained in Section 3.4.2, and values are obtained from the record file. Suppose a value is identified as an image based on the first few bytes (i.e., file signatures). In this case, optical character recognition is executed to extract characters from it, which is used for the value logging instead of the original bytes.

When the invoked method is *Runtime.exec()*, the server checks the taint of its argument register and performs the DA-flow tracking described in Section 3.3.1. All values needed for the tracking are obtained from the record file.

If the executed instruction is a TC encoder or decoder, the server examines TC flows (Section 3.3.4). The threshold should be selected depending on the environment [95]. Hence, considering 100%-accurate transmission, a threshold is determined to be 100 milliseconds by measuring the accuracy of TC flows on the device. The server performs the TC-flow tracking only when a TC encoder operates at least two different values to avoid FPs.

3.4.4 Control-Flow Analysis

The DEX bytecode provides twelve *if-** instructions for branching. The server statically analyzes control flows over a collection of the app code executed and creates control-flow graphs and dominator and post-dominator trees with the iterative algorithm [96]. Loops are controlled by the *if* and *goto-op* and are detected by finding back paths based on *goto-* and *cond-labels*, and exit paths of a loop (i.e., jumps from the loop's body to the outside) are detected [97, 98]. The results are used to detect control-dependent instructions of each loop, which are instructions within the loop's body.

When a branch is executed, the taint of its operands is checked and propagated to the destination registers of control-dependent instructions that are *unop*, *binop*, and *const-op* instructions depending on the result of the inspection explained in Section 3.3.3. VTDroid would miss flows when the attacker knows the contents of the tracked information, which will be further discussed in Section 3.6.1. In addition, the server counts and alerts control-dependent sinks for privacy leak detection because the sinks can leak information directly to a remote server.

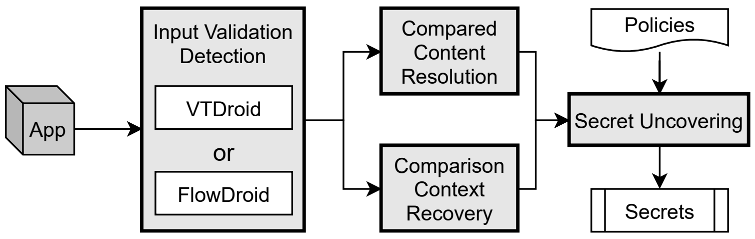

Input Validation Detection
VTDroid
or
FlowDroid
App
Compared Content Resolution
Comparison Context Recovery
Policies
Secret Uncovering
Secrets

FIGURE 3.10: Overview of InputScope in the experiment. Validations are detected by VTDroid or FlowDroid and are passed to the rest of the InputScope's components, and secrets are uncovered.

3.5 Evaluation

This section shows how VTDroid benefits security analysts compared to current tools by demonstrating the effectiveness of VTDroid in two contexts: privacy leak detection and suspicious validation detection. In privacy leak detection, taint analysis is used to detect information flows from points at which apps load sensitive information to other points at which apps send data out to the network. Taint analysis is used to detect information flows from points at which apps obtain user input to other points at which apps compare data in suspicious validation detection. The former information flows are called privacy leaks, and the latter, input validations.

3.5.1 Experiment Setup

VTDroid is executed on a machine with a ten-core 3.7GHz CPU and 128GB RAM in privacy leak detection. Apps are exercised with Zenfone 4 (Android 8.0.0, 64GB storage) device. This section involves three current trackers: FlowDroid [29], TaintDroid, and CTT [76]. FlowDroid is a state-of-the-art tool for static taint analysis. FlowDroid of version 2.8 is used. Since the evaluation focuses on taint propagation and not taint introduction/checking, FlowDroid's taint source/sink definition is upgraded when FlowDroid misses a flow because of the definition. However, FlowDroid cannot be configured to use VTDroid's taint sources for contacts and photos because the VTDroid's taint sources are value-based. Therefore, the results of FlowDroid for contacts and photos are excluded.

In the community, TaintDroid is the most stable implementation of dynamic taint analysis for Android apps, and CTT is the only currently available technique against the ATA techniques. TaintDroid (4.3r1) is used with a Nexus 4 smartphone. CTT is implemented into another Nexus 4 with TaintDroid (4.3r1). TaintDroid and CTT are modified to allow the execution of apps' native programs to increase the number of apps that the tools can analyze. It is confirmed that no FN is produced because of the modification, and the modification only positively affects the tools' results. In this experiment, each app is manually exercised on each device, games are played to complete at least one stage, accounts are created and logged in if available, and menu items are selected as many as possible. SIM cards were installed, but only WiFi was used for the communication.

InputScope [24] is an open-sourced tool for uncovering secrets from user input validations. InputScope detects user input validations by utilizing FlowDroid, which is vulnerable to ATA techniques. The environment consists of the latest version of InputScope and FlowDroid in July 2021. FlowDroid is the same version as the one used in the privacy leak evaluation. Figure 3.10 shows the overview of InputScope, consisting of four components:

input validation detection, compared content resolution, comparison context recovery, and secret uncovering. Taint analysis, FlowDroid by default, is used to detect input validations, and the validations are passed to the rest of the InputScope's components. FlowDroid is replaced with VTDroid for the input validation detection to compare two InputScopes with VTDroid and FlowDroid. After VTDroid or FlowDroid detects validations, the results are characterized by the next two components. The last component applies policies to determine behavior types: secret access key, master password, censorship behavior, and secret command.

The two InputScopes are executed on the ten-core CPU computer and exercised apps with the Zenfone 4 device. An app exerciser, Monkey, is used to exercise apps. Each app is exercised for one hour. A script is used to input texts into an app by injecting keyboard events when a keyboard is activated. The script helps Monkey to exercise apps' validation behaviors more effectively. Furthermore, apps are manually exercised when a keyboard is activated during the exercise with Monkey.

3.5.2 Taint Sources and Sinks

This section explains taint sources and sinks used by VTDroid in the evaluation.

Taint Sources

For privacy leak detection, VTDroid tracks the four categories of private information tracked by TaintDroid. It contains low-bandwidth and high-bandwidth sensors, information databases, and device identifiers. Taint is introduced to values of location and device identifiers (i.e., IMEI, IMSI, ICCID, and phone number) at the *invoke-op (API)* that calls permission-protected APIs. At the same time, the values are obtained from the record file and saved for the length tracking explained in Section 3.3.2. In contrast, the taint is introduced to values of photos and contacts at the *invoke-op (API)* based on its contents. Before the taint analysis, the server acquires photos from the device's storage and data in contacts.

For suspicious validation detection, VTDroid targets the same taint sources as InputScope. It contains three API methods that return a value of the java.lang.String or the android.text.Editable.

Taint Sinks

For privacy leak detection, the server checks the taint at invocations of API methods relating to streams of binary, objects, bytes, and characters that are DataOutput, ObjectOutput, OutputStream, Writer, ByteArrayInputStream, and their subclasses, 32 classes in total. Note that the classes should not always cause a leak, and the presented sink definition is considered out of scope. VTDroid also covers constructors of common HTTP-communication classes that are java.net.URL, okhttp3, org.apache.http, and com.android.volley. For suspicious validation detection, VTDroid checks the taint for 11 API methods that are InputScope's taint sinks [24].

3.5.3 Datasets

Two test suites were created to evaluate VTDroid and the current tools with apps actively utilizing ATA techniques. As test suites, 34 ATA techniques in Table 3.1 are implemented, and 31 distinct techniques are obtained (there are eight TC techniques in Table 3.1, but five

TABLE 3.4: Number of techniques detected by the four privacy-leak detectors in the test suite.

System	ATA Techniques DA	MO	CD	TC	Sum
VTDroid	5	5	16	5	31
CTT	2	3	2	1	8
TaintDroid	0	0	0	0	0
FlowDroid	0	1	0	0	1
#Techniques	5	5	16	5	31

TABLE 3.5: Number of apps alerted in the popular apps. The column "Any" gives the number of apps leaking at least one source. Note that "Any" for FlowDroid excludes contacts and photos.

System	Any	IMEI	IMSI	ICCID	Location	Contacts	Photos	Phone Number
TPs / FPs / FNs								
VTDroid	11 / 1 / 0	11 / 0 / 0	3 / 1 / 0	1 / 0 / 0	0 / 0 / 0	0 / 3 / 0	0 / 1 / 0	0 / 0 / 0
CTT	11 / 12 / 0	11 / 8 / 0	3 / 11 / 0	1 / 2 / 0	0 / 5 / 0	0 / 4 / 0	0 / 1 / 0	0 / 2 / 0
TaintDroid	9 / 0 / 2	9 / 0 / 2	0 / 0 / 3	1 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
FlowDroid	3 / 0 / 8	3 / 0 / 8	0 / 0 / 3	0 / 0 / 0	0 / 0 / 0	- / - / -	- / - / -	0 / 0 / 0
Precision / Recall								
VTDroid	0.92 / 1.00	1.00 / 1.00	0.75 / 1.00	1.00 / 1.00	- / -	0.00 / -	0.00 / -	- / -
CTT	0.48 / 1.00	0.58 / 1.00	0.21 / 1.00	0.33 / 1.00	0.00 / -	0.00 / -	0.00 / -	0.00 / -
TaintDroid	1.00 / 0.82	1.00 / 0.82	- / 0.00	1.00 / 1.00	- / -	- / -	- / -	- / -
FlowDroid	1.00 / 0.27	1.00 / 0.27	- / 0.00	- / -	- / -	- / -	- / -	- / -

implemented TC techniques represent them). The test suite for privacy leak detection contains 31 ATA techniques. Each technique accesses IMEI and sends the data out to the network. Note that the Direct Buffer and Lookup Table tricks are replaced with the ATA techniques in Figure 3.3. The other test suite for suspicious validation detection contains the same ATA techniques except for three techniques: Bitmap-Cache-based DA and CD flows and the Remote-Control-based TC flow. The three techniques were excluded because their destination of information transfer is a remote server. Any validation outside of a device is out of the scope of InputScope. Each technique in the test suite obtains user input and compares the data with a hardcoded string. This behavior is categorized as master password based on the policy [24].

VTDroid and the current tools were also evaluated with popular real-world apps, which contain more complex code than apps in the test suites and are less likely to operate ATA techniques. Recently-published apps are not suitable for privacy leak detection because they do not access TaintDroid's taint sources, as obtaining such sensitive information is restricted year by year. Therefore, 30 apps were randomly chosen from the most popular apps that were collected from the Google Play Store in 2016 and are shown to be more complex than non-popular apps [21]. For suspicious validation detection, recently published apps were collected from the Google Play Store and Baidu app store in July 2021. The top 200 free apps and the top 200 free games on the Google Play Store were downloaded. Then, Android App Bundles were omitted, and finally, 277 apk files were obtained. Also, apps were downloaded from five rankings on the Baidu Store, duplicates and Android App Bundles were omitted, and finally, 226 apk files were obtained. Complete lists of hashes of the apps mentioned here are given in Appendix A.

3.5.4 Privacy Leak Detection

This section discusses the results of privacy leak detection.

Test Suite

The four systems analyzed the 31 ATA techniques (Table 3.4) in the test suite. TaintDroid detects no technique. FlowDroid detects an MO flow with the taint rules for *Bitmap.setText()* and *Bitmap.getText()*. CTT detects eight more techniques than TaintDroid. In contrast, VT-Droid successfully detects all the techniques. The value logging and matching, not relying on API method names, successfully detected all the non-data-transforming DA flows, which are the Text Scaling, Bitmap Cache, and File Last Modified, and their invisible CD flows, six techniques in total. The results demonstrate that VTDroid is more effective than the other tools for analyzing apps aggressively exploiting the ATA techniques.

Popular Apps

Table 3.5 compares the results of the experiment with the 30 popular apps. TPs indicate actual leak-causing apps detected by a system. FPs represent no-leak-causing apps falsely flagged by a system. FNs show apps accomplishing leaks without triggering an alarm by a system but detected by the others. The ground truth is obtained by verifying the detected leaks based on the captured HTTP/HTTPS traffic.

VTDroid and CTT generate no FNs, while TaintDroid fails to detect two and three leaks of IMEI and IMSI, and FlowDroid misses eight and three leaks of IMEI and IMSI detected by the others. It was found that VTDroid detects information flows of the five leaks with rules for MO flows (Section 3.3.2) and the DEX bytecode instructions (Section 3.4.3). It was also confirmed that none of the DA, CD, and TC flows is involved in the information flows of all leaks. There is no difference between the numbers of TPs detected by VTDroid and CTT. Note that for fairness, leaks caused only on certain devices were manually identified and omitted. Hence, security analysts can use either VTDroid or CTT to avoid missing leaks.

However, there are noticeable differences between the numbers of FPs detected by the tools, especially for apps leaking IMEI, IMSI, or location. Overall, VTDroid triggers only one more FP than TaintDroid and FlowDroid, whereas CTT produces 12 more FPs than TaintDroid and FlowDroid (indicated by the column “Any”). The results indicate that a CTT user should verify 12 apps that leak no information, while a VTDroid user should check only one no-leak-causing app.

Although both VTDroid and CTT produce certain numbers of FPs, there is a significant difference between VTDroid and CTT. Figure 3.11 shows the amounts of false alarms produced by the systems with apps that leak no sensitive information. The numbers of executed sinks are almost the same among the systems: 913 sinks were executed during VTDroid’s analysis; 789, CTT’s analysis; and 738, TaintDroid’s analysis. In contrast, the amount of sinks falsely alarmed by CTT is notably high compared to VTDroid and TaintDroid. CTT generated 541 FPs, while VTDroid and TaintDroid produced 20 and 0 FPs, respectively. The FP ratio is 69% for CTT and 2% for VTDroid. TaintDroid is 100% accurate. The difference is because CTT taints the return value of API methods that do not usually cause information flows, as Section 3.1.2 points out. In the situation that an analyst attempts to verify that an app causes no leak, all the app’s alarms produced by a tool must be confirmed to be false. Thus, the number of false alarms significantly affects the analysis cost. The overall results

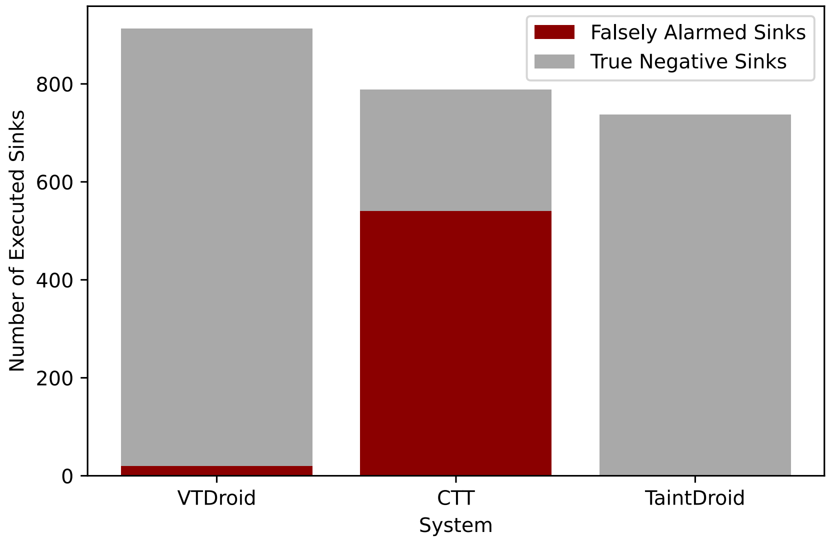


FIGURE 3.11: Ratio of false alarms to the number of executed sinks in apps that leak no sensitive information. FlowDroid is excluded because FlowDroid performs static analysis and cannot count the sink execution number.

TABLE 3.6: Number of techniques detected by the two suspicious-validation detectors in the test suite.

System	ATA Techniques				Sum
	DA	MO	CD	TC	
VTDroid	4	5	15	4	28
FlowDroid	0	0	0	0	0
#Techniques	4	5	15	4	28

demonstrate that VTDroid is more effective than CTT, TaintDroid, and FlowDroid for analysts who do not tolerate FPs and FNs.

3.5.5 Suspicious Validation Detection

This section presents the results of suspicious validation detection with popular apps. The experiment was to evaluate the increase in the number of taint propagation and its verification cost due to the taint propagation rules with the current tool as the basis.

Test Suite

The InputScopes with VTDroid and FlowDroid analyzed the 28 ATA techniques in the test suite (Table 3.6). InputScope with VTDroid successfully detects all the techniques and uncovers the master password secret. In contrast, InputScope with FlowDroid identifies none of the techniques and fails to uncover any suspicious validation. The reason why FlowDroid misses the MO flow, detected by FlowDroid in the privacy leak evaluation in Table 3.4, is possibly the FlowDroid's options set by InputScope. The results show that VTDroid is more effective than FlowDroid for analyzing apps conducting the ATA techniques.

TABLE 3.7: Results of suspicious validation detection by InputScopes with VTDroid and FlowDroid. The values in parentheses show the number of apps. Validations and secrets detected by VTDroid (shown in the columns Total) contain all the validations and secrets detected by FlowDroid within the same coverage. Validations and secrets detected only by VTDroid are shown in the columns Only. C. stands for coverage, and the numbers within the VTDroid's and full coverages are shown for FlowDroid.

Dataset	**#Apps**	**#Validations**				**#Secrets**			
		VTDroid		**FlowDroid**		**VTDroid**		**FlowDroid**	
		Only	**Total**	**VTDroid's C.**	**Full C.**	**Only**	**Total**	**VTDroid's C.**	**Full C.**
Google	277	150 (6)	158 (9)	8 (3)	312 (60)	6 (2)	9 (3)	3 (1)	37 (14)
Baidu	226	443 (2)	457 (6)	14 (4)	3,201 (119)	0 (0)	0 (0)	0 (0)	40 (17)
Sum	503	593 (8)	615 (15)	22 (7)	3,513 (179)	6 (2)	9 (3)	3 (1)	77 (31)

TABLE 3.8: Number of taint propagation by the taint propagation rules and legacy rules in suspicious validation detection, and coverage of the apps' code. The values in parentheses indicate the ratio to the total.

Dataset	**App**	**Total**	**DA**	**MO**	**CD**	**TC**	**Legacy**	**Coverage**
Google	1	1,369,890	97,505 (7.1)	260,298 (19.0)	107,888 (7.9)	43,395 (3.2)	860,804 (62.8)	13.7
	2	821,544	15,976 (1.9)	348,046 (42.4)	114,233 (13.9)	324 (0.0)	342,965 (41.7)	25.5
	3	174,226	3,555 (2.0)	44,356 (25.5)	12,072 (6.9)	1,289 (0.7)	112,954 (64.8)	38.6
	4	148,022	1,363 (0.9)	82,804 (55.9)	1,393 (0.9)	2,114 (1.4)	60,348 (40.8)	7.8
	5	126,304	674 (0.5)	76,286 (60.4)	11,262 (8.9)	0 (0.0)	38,082 (30.2)	5.7
	6	100,778	16,223 (16.1)	18,334 (18.2)	18,958 (18.8)	34 (0.0)	47,229 (46.9)	34.2
	7	79,474	394 (0.5)	12,176 (15.3)	2,311 (2.9)	0 (0.0)	64,593 (81.3)	22.0
	8	77,879	13,831 (17.8)	10,445 (13.4)	24,266 (31.2)	33 (0.0)	29,304 (37.6)	27.6
	9	59,944	7,829 (13.1)	22,015 (36.7)	6,491 (10.8)	1,004 (1.7)	22,605 (37.7)	6.7
Baidu	10	694,610	18,763 (2.7)	254,837 (36.7)	28,238 (4.1)	35,438 (5.1)	357,334 (51.4)	16.5
	11	597,006	88,934 (14.9)	247,311 (41.4)	60,806 (10.2)	22,970 (3.8)	176,985 (29.6)	5.2
	12	523,375	34,217 (6.5)	212,433 (40.6)	32,671 (6.2)	9,931 (1.9)	234,123 (44.7)	20.9
	13	483,953	15,375 (3.2)	259,219 (53.6)	32,007 (6.6)	19,115 (3.9)	158,237 (32.7)	8.5
	14	21,870	2,978 (13.6)	11,995 (54.8)	1,086 (5.0)	1 (0.0)	5,810 (26.6)	4.7
	15	7,930	272 (3.4)	3,344 (42.2)	732 (9.2)	0 (0.0)	3,582 (45.2)	5.3

Popular Apps

Table 3.7 shows the numbers of apps, validations, and secrets detected by InputScopes with VTDroid and FlowDroid. InputScope with FlowDroid uncovers secrets in 31 apps among 503 apps in total (6%), which is almost the same rate as the original paper's results. In comparison, VTDroid detects a smaller number of validations. The difference is because FlowDroid statically analyzes the entire code of apps (i.e., full coverage). VTDroid dynamically analyzes only the executed code of apps, and code coverages are different between the two analyses. Code coverage of VTDroid is 70.35% at the highest, 9.41% on average, and 0.04% at the lowest. It was confirmed that validations detected only by FlowDroid were not executed during the VTDroid's analysis.

It is noteworthy that FlowDroid detects a few validations within the same coverage of VTDroid. VTDroid detects 150 validations (six apps) and 443 validations (two apps) in the datasets that FlowDroid overlooks. The differences are caused by the difference in taint propagation rules employed by VTDroid and FlowDroid. VTDroid employs the taint propagation rules for DA, MO, CD, and TC and other legacy rules that are for direct-assignment bytecode instructions, including *const-op*, *move-op*, *unop*, *binop*, *iget-op*, *iput-op*, *sget-op*, *sput-op*, and *move-result-op* of *invoke-op (non-API)*. In contrast, FlowDroid employs only the legacy rules and is not able to track flows of the four types as demonstrated in Section 3.5.5. Table 3.8 shows the numbers of taint propagation by the rules and legacy rules with the 15 apps that VTDroid detects the validations shown in Table 3.7. Each value indicates the number of increases of tainted registers calculated by counting the execution of each rule with a tainted register as its source operand of propagation. Note that the rule for *const-op* only removes the taint of its destination operand, and the number of cleared taints by the rule was counted. The data shows that information flows of any app are tracked by the rules for DA, MO, CD, and/or TC, which FlowDroid does not employ. Thus, FlowDroid fails to track some information flows in any app listed in Table 3.8 and could miss secrets if apps performed suspicious validations with the flows. Hence, VTDroid is more effective than FlowDroid for analysts who require a robust approach against the ATA techniques.

There is a trade-off between robustness and precision, and a robust tool could not be practically used if the tool generates numerous FPs. Among the 615 validations detected by VTDroid, only nine validations (three apps), shown in column 8 from Table 3.7, are flagged as secrets by the last component with the policies. The latter three components of InputScope screened out the other 606 validations. Note that it does not mean the 606 validations are all false positives, but they are simply excluded from the final outcome because their compared contents or comparison contexts are not suspicious. The nine validations were verified, and three of them are TPs. FlowDroid also detects them (column 9). The other six validations are tracked by the rules for DA, MO, and CD flows. Three of them are detected with a DA flow that consists of an invocation of *java.lang.CharSequence.charAt()* API method as either transmitter or receptor. The official Android API reference shows that the API method returns a character from its character sequence (i.e., base object). The API method only causes a flow from the base object to the return value, which means it can be neither transmitter nor receptor, and the detected DA flows do not actually exist. Hence, the three secrets were identified as FPs. The other three validations are detected with a TC flow. The bytecode of the TC flow's encoder was manually analyzed, and it was found that the encoder is an *if* instruction checking whether the taint source value is null immediately after the value is obtained (i.e., the encoder cannot preserve information of the taint source). It can be concluded that the TC flow is FP. The verification cost is negligible, and it can be considered that VTDroid achieves both robustness and precision.

3.5.6 Performance Evaluation

The performance of the tools used in the privacy leak evaluation will be discussed separately for static and dynamic analyses. FlowDroid, the only static analysis tool in the privacy leak evaluation, completed the analysis in 133 minutes, which is slightly longer than CTT and TaintDroid. FlowDroid analyzed the apps one by one. The time can be easily shortened by executing the analysis parallelly if more computation resources are available. However, FlowDroid results in the least recall in Table 3.5, meaning that FlowDroid misses more flows than the other tools.

On the other hand, the analysis time depends on the app exercise time for the dynamic analysis tools. Since apps were manually exercised in privacy leak detection, an app's exercise time increases roughly in proportion to the number of the app's features. To finish exercising all the popular apps on the devices, VTDroid took 101 minutes; CTT, 96 minutes; and TaintDroid, 86 minutes. Since VTDroid instruments code before launching apps, it increases the size of the app, the installation time, and the app runtime overhead. The app runtime overhead is caused by the instrumented code writing logs to the device's storage. In exchange, VTDroid users can use the recorded execution traces to understand the app's

```
/url?pkg=<value>&ver=1011&ha=0a&lc=en_US
```

FIGURE 3.12: An example of HTTP request containing transformed sensitive information.

behaviors, while TaintDroid and CTT do not record such information in the experiment. If TaintDroid or CTT were modified to record the information, the tool would write logs to storage, which is the same type of operation as VTDroid's instrumented code, and the tools' app runtime overheads would be close to each other. Therefore, the app runtime overhead of VTDroid is reasonable for researchers regularly requiring the information of execution traces to understand apps' suspicious behaviors. In the privacy leak evaluation, for example, it was found that an app sends an HTTP request to a server in Figure 3.12, which apparently not containing sensitive information. However, VTDroid flags it as a leak and shows a history of taint propagation with actual values processed from a source to the sink. The history indicates that characters *0a* are the first two bytes of an encoded string compounding IMEI, device name, and WiFi MAC address. In contrast, the other tools detect the leak and do not describe it, and it is infeasible for researchers to verify whether the detection is true.

Performances of the core data flow tracker cannot be evaluated only by comparing the app runtime overheads. This is because TaintDroid and CTT implement taint analysis on Android devices, while VTDroid's taint analysis is implemented on the server. VTDroid also takes time for the server-side taint analysis. With the 30 popular apps in privacy leak detection, the time taken to complete the analysis per app was calculated (Figure 3.13). A timeout of one hour was used per analysis. VTDroid analyzes an app with tracking a single private information per analysis, and the time varies depending on the tracked information. For each app, the maximum and minimum show the analysis times of the longest-time-taken and the shortest-time-taken private information, respectively. Since multiple private information can be tracked in parallel, the analysis time for an app is determined by the maximum. The results show that the analysis finishes within three minutes at a maximum for 67% of the apps and within ten minutes for 87% of them. Code coverage was measured, and the highest is 43%, the average is 15%, and the lowest is 4%. As an example of parallel-analysis time, VTDroid was executed to analyze the 30 apps one by one on the condition that VTDroid tracks the seven private information at the same time. VTDroid was finished in 219 minutes. Overall, VTDroid took 29 minutes for the static bytecode instrumentation, 101 minutes for the app exercising, and 219 minutes for the information flow analysis, and the total is 349 minutes. It is the longest among the tools but is still a few hours, which poses no problem for practical use. The results demonstrate that VTDroid is practical for analyzing popular real-world apps.

In suspicious validation detection, each app was exercised for one hour, and a timeout of one hour was used for VTDroid. In addition, VTDroid takes several minutes for the static bytecode instrumentation per app. Hence, a timeout of three hours was used for FlowDroid for a fair comparison. Because of the timeouts, both VTDroid and FlowDroid failed to analyze some apps. Many factors can cause the analysis failure, for example, code complexity and buggy apps. Improving the applicability of the tools is out of the scope of this paper, and the results of successfully analyzed apps were focused.

Since the analysis times for both VTDroid and FlowDroid depend on the timeouts in this experiment, the performance of the tools for the apps whose analysis results do not

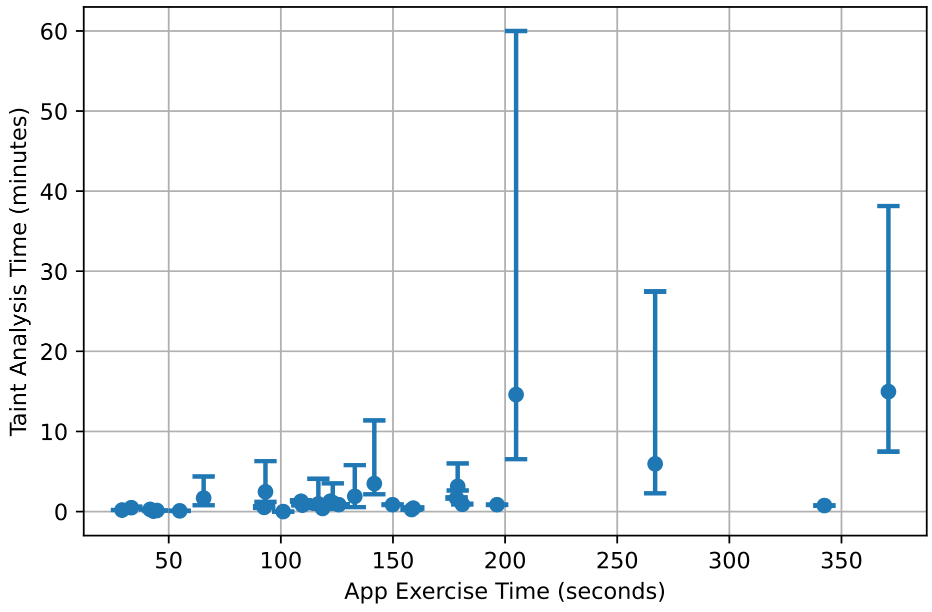


FIGURE 3.13: Relation between app-exercise and server-side analysis times per app. For each app, the upper and lower whiskers and dot denote the maximum, minimum, and average of seven analysis times taken to track each of the seven private information.

match cannot be compared. For three apps from the Google Play Store and four apps from the Baidu app store that the validation detection results of VTDroid and FlowDroid are the same, the analysis time for VTDroid was 560 minutes and for FlowDroid, 12 minutes. The result shows that FlowDroid is faster than VTDroid. A major reason is that VTDroid takes time to exercise the apps, which is necessary to obtain the apps' runtime values. The time can be shortened by improving the exerciser.

3.5.7 Ethical Considerations

The experiments were carried out only by the author and his supervisors, and no actual personal information was involved. The respect for app publishers should be considered, and the instrumentation of the app code is designed to not change any data sent to remote servers, not to affect their properties. In addition, the results were carefully used only for the purpose of the experiments and not used with any other intention.

3.6 Discussion

This section explains the robustness of the taint propagation rules and the limitations of VTDroid.

3.6.1 Robustness of Taint Propagation

The tracker for both data-transforming and non-data-transforming DA flows performs taint propagation based on value matches. Thus, the tracker generates FPs when an API method is executed with a value that coincidentally matches one of the unrelated transmitter's previously logged values. It is shown that the tracker generates fewer FPs than CTT in the privacy

```
1  x = "passwd";
2  TextView tv1 = findViewById(R.id.a);
3  TextView tv2 = findViewById(R.id.a);
4  tv2.setTextScaleX(0);
5  if (x == "topSecret") {
6      tv1.setTextScaleX(1);
7  }
8  y = tv2.getTextScaleX();
```

FIGURE 3.14: Combination of invisible CD and hidden implicit flows $x \Rightarrow y$ w/ class.

leak evaluation and that the FP verification cost is small in the input validation evaluation. Therefore, the impact of FP caused by the tracker is negligible.

The value logging and matching are based on the one-value feature, which assumes that transferred values are not changed through unmonitored memory. However, this assumption could be incorrect depending on API methods. Also, VTDroid tracks values printed in an image with optical character recognition, but more techniques are available to hide data in an image. Failing to track DA flows with the value logging and matching can cause failures in tracking the other types of flows. The invisible-CD-flow tracking (Section 3.3.3) and TC-flow tracking (Section 3.3.4) utilize the result of DA-flow tracking. Hence, if VTDroid misses a DA flow, it fails to detect CD and TC flows associated with the DA flow. Therefore, conducting further investigation and characterization of API methods considering data-transformation capability is needed. Since VTDroid users can quickly test their taint propagation rules on the analysis server without running apps repeatedly, it is believed that this study will serve as a base for future research on taint analysis of Android apps.

The shell command parser explained in Section 3.4.2 currently considers pipes, parentheses, and redirect operators as delimiters to separate shell commands in ATA techniques in the test suite. It was manually confirmed that the parser properly functioned with shell commands (e.g., *getprop* and *type*) executed by the popular apps in the experiments. A more well-formalized parser should be utilized in the future. However, the parser will still be limited to shell commands and cannot track flows in external programs, such as Python scripts.

A limitation of the CD-flow tracking is to track a CD flow that has not one-to-one but many-to-one mapping, which can preserve not a perfect but a certain amount of information. In addition, the CD-flow tracking does not detect a combination of the defined invisible CD and hidden implicit flows [99]. Figure 3.14 shows an example of the combinational flows. Since x's value is *passwd*, the branch at line 5 is not taken. Subsequently, the transmitter at line 6 is not executed, and the value logging and matching do not detect the flow between lines 6 and 8. In this case, however, the branch only transfers information that x's value is not *topSecret*. As long as the branch is not taken, such comparisons with predicted values are considered as non-information-preserving because values of device identifiers and user input are not predictable. In addition, such comparisons are narrow conditions, which must be detected as suspicious by other methods such as symbolic execution. Therefore, VTDroid is currently allowed to miss the combinational flows. Also, the CD-flow tracking is currently the only practical solution because current implicit flow tracking, such as [100, 101], taints destination registers of control-dependent instructions when a branch is not taken, and no register is tainted in the example. Therefore, the CD-flow tracking is acceptable and practical.

The TC-flow-tracking threshold value used in the evaluation was determined by the authors for the experiment environment. Since the balance between FP and FN changes depending on the threshold value, the user must decide according to the balance the user desires. Since the TC-flow tracking performs based on the execution time of API methods, it is considered that low-level elements in API methods do not affect the tracker. Stephens et al. [102] develop a program obfuscation framework that leverages several channels exploiting low-level elements, such as just-in-time compilation and garbage collection, to encode data into timing. However, their target is not Android, and it is unclear whether their channels work in the Android platform.

3.6.2 Limitations

Code coverage depends on app-exercising methods and experiment environments in dynamic analysis. Thus, the VTDroid's coverage in Section 3.5 does not perfectly represent code executed on real users' devices. Scalability was prioritized, and Monkey was mainly used in the suspicious validation detection. By improving the coverage, VTDroid should detect the validations uncovered by FlowDroid within full coverage (Table 3.8) except FlowDroid's FPs. Progressing the app-exercising methods and experiment environments is out of this paper's scope, and the coverage should be improved by current approaches [103, 104] focusing on analysis environments.

VTDroid currently does not track information flows between multiple apps. However, API methods provide shared resources to apps, which can be used for covert channels [105, 106]. VTDroid cannot track data leaving and returning to a device in a network reply because a remote server can modify it into any format. Also, native code, such as third-party libraries, is out of focus of VTDroid as well as bytecode-level trackers. They are written by app developers and packaged within apps and are different from API methods, not written by third parties. Since it is easy to locate such libraries, they should be analyzed by tools such as Malton [70].

Since the approach uses image data to introduce taint to images, analysts need to prepare images carefully by themselves if they want to detect image leaks. Preparing images is out of the scope of this paper. When the analysis server retrieves images from the Android device, it would take time and consume a large disk space if there are many images. Also, if there are few images on the device, apps may detect a small number of images and hide their behavior.

3.7 Related Work

This section discusses related work. Until now, VTDroid is the only tracker that targets the combinations of the four flow types. Therefore, this section presents current approaches for each of the DA, MO, CD, and TC flows.

3.7.1 DA Flow Tracking

Similar to the value logging and matching, content-based tracking [93] is devised to track information flows over the unmonitored area. However, the approach cannot track the other types of flow and does not target the Android platform. Although full-system taint trackers are developed [70, 67], they do not track control dependencies and can miss flows on the runtime layer. For example, the major purpose of Malton [70] is to provide a comprehensive view of the target apps. As such, it is not designed to mitigate undertainting on the runtime layer and across API calls.

3.7.2 MO Flow Tracking

TaintDroid tracks two MO flows of the Direct Buffer and Lookup Table. However, the Direct Buffer can use a tainted value as the location (Figure 3.3) to transfer information without being tracked by TaintDroid. TaintDroid tracks the Lookup Table with a propagation rule for the *aget-op* because the technique is commonly used for character conversion. However, this paper presented an example (flow $x \Rightarrow y1$ in Figure 3.3), which TaintDroid will fail to track. It was shown that one of the rules, Rule 4, can track the flow, and the rule was also tested with real-world apps. Other than TaintDroid, taint propagation rules for the array put operation are previously presented by Kynoid [107], TaintART [62], MirrorDroid [90], and TaintMan [64]. However, none of their rules propagates the taint from the index to the array (Rule 4 in Table 3.2), and they do not track CD flows. Hence, their approaches miss the information flow. An app-level taint tracker [65] propagates taint among reference-data-type arguments, base objects, and return values of API methods. Therefore, it could track some invisible MO flows. However, it triggers FNs for methods taking primitive-data-type arguments. Also, tainting return values of length-related methods can trigger overtainting with CD flows because branches often operate lengths.

3.7.3 CD Flow Tracking

You et al. [64] developed CD-flow detection that checks the values of registers at re-convergence points. However, it will fail when information is transferred to API calls, not to registers, which cannot be examined at the re-convergence points. Researchers devised CD-flow tracking techniques [100] for TaintDroid and [108] for FlowDroid. FlowDroid also provides the option of tracking CD flows and could detect some ATA techniques in the test suite. Even though, unlike the CD-flow tracking, they only focus on visible CD flows and cannot track invisible CD flows and the other types of flow.

3.7.4 TC Flow Tracking

Researchers developed techniques that detect timing channels by analyzing API usage frequencies [86] or the time differences between send operations [87]. Nevertheless, their approaches do not track information flows within apps and have limited uses. For example, the techniques cannot be utilized for input validation detection. Until now, CTT is the only approach to tracking information flows over TC flows. However, tracking only TC flows on the runtime layer will fail to detect TC flows in the shell commands. Detecting information flows between an app and shell commands is essential to track such flows, which the TC-flow tracker achieves.

3.8 Summary

This chapter presented a new approach to tracking visible and invisible information flows originating from ATA techniques. The characterization of the flows enables a tracker to detect all the ATA techniques in ScrubDroid and other tricks utilizing unknown media in API calls. A taint tracker called VTDroid was built for privacy leak and user input validation detections, and its effectiveness was evaluated compared to TaintDroid, CTT, and FlowDroid. The results show that VTDroid outperforms the current trackers. VTDroid's performance demonstrated in this chapter should be an indicator for researchers to determine whether they are concerned with ATA in their studies.

Chapter 4

T-Recs: Tracking Information Flows by Recording and Reconstruction

Protecting smartphone users has attracted increasing interest due to the appearance of suspicious apps and third-party SDKs. App analysts apply automatic analysis techniques to large-scale datasets of Android apps to detect suspicious behaviors. For example, Zhao et al. uncovered backdoor and blocklist secrets in apps published on the Google Play Store and Baidu app store and pre-installed apps [24]. They used a static taint analysis tool called FlowDroid [29] to uncover the secrets. Static taint analysis is popular in analyzing a large-scale dataset because of its scalability.

However, static taint analysis has the problem of detecting incorrect execution paths, increasing the cost of verifying experiment results. Recent reviews of the literature on static taint analysis showed the limitations of the analysis [77, 78]. Zhang et al. evaluated currently-available static taint analysis tools: FlowDroid, Amandroid [31], and DroidSafe [30] with DroidBench [79] apps supported by the tools and real-world apps. The results show that the tools are inaccurate and cannot be used for analyzing real-world apps dependably. Handling inter-component communication (ICC), reflective calls, and component lifecycles are significant challenges. For example, FlowDroid supports ICC detection with IccTA [32] and IC3 [33], which are shown to be unreliable. The increase in false positives (FPs) complicates analysis-result verification, causing an increase in analysis cost. For example, Zhao et al. manually analyzed 70 out of over 16,000 detected apps to estimate the accuracy, and the result is 87.14% (i.e., nine apps are FPs) [24]. Three of the FPs were caused by conflicting constraints along the execution path. Such a manual analysis, especially finding path constraints, is complex and requires significant effort. Also, increasing the complexity of the analysis algorithms increases the analysis time, precluding the analysis from completing in a reasonable time.

On the other hand, a dynamic taint analysis only analyzes the executed paths, and there is no chance of detecting incorrect execution paths. For example, TaintDroid [54] generates no FP in the privacy leak evaluation with 30 popular real-world apps. However, current dynamic taint analysis tools depend on specific devices and versions of Android OS. It decreases their usability [80] and the range of analyzable apps. In addition, the app exercise must be performed every time the taint analysis runs. Therefore, when the analyst or researcher changes the parameters or features of the taint analysis and re-analyzes the same app, the app exercise also needs to be executed, which incurs extra costs.

This chapter presents a new runtime-data-utilized taint analysis system called T-Recs [82], with which users can start analyzing apps immediately after plugging an unmodified device into their computer. T-Recs first records the target app's runtime information at almost instruction-by-instruction. Then, it accurately reconstructs the app execution on the computer with the logged information to track information flows and detect information leaks, whereas overcoming the device dependency issue. The computer is outside the Android framework in contrast to TaintDroid, which is implemented on the Android framework. Also, the logs are stored in the analysis computer so that the taint analysis can be executed independently of the app exercise.

T-Recs consists of five components: parser, instrumentator, logger, reconstructor, and exerciser. The parser, the instrumentator, and the logger procure the app's runtime information with as minimal bytecode instrumentation as possible. Then, the reconstructor reproduces the app execution based on the parsed and logged information. The exerciser addresses how to trigger the target behavior in apps, a general challenge in dynamic analyses. App exercise depends on the apps and the data to be tracked. For example, tracking user inputs requires input-related exercises. The exerciser triggers ICC, callbacks, and lifecycles in the DroidBench apps because the static taint analysis has limitations for handling them.

T-Recs' accuracy, analysis time, and success rate were evaluated in privacy leak detection compared to currently available taint analyzers, which are FlowDroid (w/ and w/o IC3), Amandroid, DroidSafe, DroidRA [40, 41] (w/ FlowDroid, Amandroid, and DroidSafe), IccTA, and TaintDroid (w/ and w/o IntelliDroid [59]). The results show that T-Recs outperforms the compared tools in detection accuracy. T-Recs also achieves reasonable analysis time and success rate. Further, T-Recs detects ICC- and reflection-related leaks missed by FlowDroid in popular apps collected from the Google Play Store in 2016 and 2021. For identifying and counting the leaks, a debugging feature was added to the reconstructor. Then, only the reconstructor was executed. These experiments were conducted once the overall analysis was finished, indicating that T-Recs' taint analysis (i.e., the reconstructor) can be re-executed without re-exercising the app, which is one of T-Recs' advantages. T-Recs' app-runtime overhead (i.e., overhead for apps to be installed, launch, cause leaks, and be uninstalled) and parallel execution performance were also evaluated in comparison with the other trackers. The results are acceptable, and running T-Recs in parallel can easily shorten the analysis time. T-Recs has been made available to the community.

Here is the summary of the contributions:

- A mechanism for accurately reconstructing the app execution outside the Android framework based on the app's runtime information logged on a device was developed.
- The mechanism was implemented into a new runtime-data-utilized taint analysis system called T-Recs with nearly 17,000 lines of Python and Smali code.
- It was demonstrated that T-Recs' leak detection performance compared to FlowDroid, Amandroid, DroidSafe, DroidRA, IccTA, TaintDroid, and IntelliDroid with DroidBench, 254 popular apps from the Google Play Store in 2016 and 2021, and SDK-version-varied apps from the Google Play Store and Anzhi [83].
- The importance of tracking ICC- and reflection-related flows is highlighted by T-Recs, detecting these flows in ten apps and related leaks in six apps among 96 apps from the Google Play Store in 2016 and also these flows in 52 apps and associated leaks in 29 apps among 158 apps from the Google Play Store in 2021.

```
 1  .class LLeaker;
 2
 3  imei:Ljava/lang/String;  // field
 4
 5  .method public constructor <init>()V
 6
 7      invoke-direct {p0}, Landroid/app/Application;-><init>()V
 8  .end method
 9
10  .method public callback1()V
11
12      invoke-virtual {v0}, Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/String;  // source
13
14      move-result-object v1  // tainted
15
16      if-eqz v1, cond_0
17
18      iput-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field setter
19
20      :cond_0
21  .end method
22
23  .method public callback2()V
24
25      iget-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field getter
26
27      invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I  // sink
28  .end method
```

FIGURE 4.1: Smali code leaking the sensitive information.

- The additional experiments indicate that T-Recs's cost of re-executing taint analysis is small, taking 34 minutes (17% of the whole) for the 96 apps collected in 2016 and one hour and 40 minutes (11% of the total) for the 158 apps collected in 2021.

The rest of this chapter is organized as follows. Information leaks and taint analysis are explained in Section 4.1. The approach is presented in Section 4.2, and its implementation is described in Section 4.3. The evaluation is reported in Section 4.4, and the results are discussed in Section 4.5. Related work is explained in Section 4.6. Lastly, the summary is stated in Section 4.7.

4.1 Background

This section first discusses an information-leaking app's code. Then, it explains information flow tracking and current taint analysis approaches.

4.1.1 Information-Leaking App's Code

An app can leak sensitive information with, for example, the Smali code in Figure 4.1. In Smali, one class is defined per file, similar to Java. The class name is printed in the first line of the Smali file. It is prefixed with a capital *L* and suffixed with a semicolon. In Figure 4.1, *Leaker* is the class name. A field of *Leaker* is defined in Line 3. The left part of the colon is the field name (i.e., *imei*), and the right part is the data type (i.e., *Ljava/lang/String;*). The data type is *java.lang.String* in Java, and slashes are used instead of dots in Smali.

Leaker has three methods defined in Lines 5-28. A method definition begins from a line starting with *.method* to next *.end method* line. A constructor is defined in Lines 5-8; *callback1()*, 10-21; and *callback2()*, 23-28. The *V* attached to the method names' ends indicates the data type of the return value, and *V* means void in Java. The *invoke* instructions, such as *invoke-direct* in Line 7, *invoke-virtual* in Line 12, and *invoke-static* in Line 27, are used to call a method. The *move-result-object* instruction assigns the return value of the most recent *invoke* instruction to the destination register (e.g., *v1* in Line 14). The *if-eqz* instruction is one of the branch instructions, and the path to the label is taken if the operand register value is 0 (e.g., if *v1*'s value is 0 in Line 16, Line 18 is skipped, and Line 20 is subsequently executed). The *iput-object* and *iget-object* instructions in Lines 18 and 25 are a setter and a getter of instance fields, respectively.

4.1.2 Information Flow Tracking

The information flow starts when the code obtains a device's hardware identifier (i.e., IMEI) in Lines 12-14 in the method *callback1()*. The code sets the value to the field *imei* in Line 18. Then, the code moves the value from the field *imei* to the register *v1* in Line 25 and leaks it by calling *Log.i()* in Line 27 in the other method *callback2()*.

Assume that *callback1()* is called, a taint tracker can assign a taint tag to the register *v1* in *callback1()* and propagates the taint from *v1* to the field *imei*. A challenge is to determine whether *callback1()* and *callback2()* are executed in this order. Execution of the methods depends on ICC, user-interface events, and the app's lifecycle. If a tracker overapproximates the call flows, the leak is falsely detected (i.e., FP). Alternatively, if a tracker underapproximates, the leak is missed (i.e., false-negative (FN)).

4.1.3 Static Taint Analysis

Static taint analysis requires no Android device and processes apps without running them. A significant challenge is to obtain precise Android models to find correct execution paths (e.g., the execution order of *callback1()* and *callback2()* in Figure 4.1). Also, the execution order of instructions changes based on system properties, such as OS version and IMEI (e.g., Line 16 in Figure 4.1).

Considerable effort has been devoted to Android-modeling techniques; however, the limitations are demonstrated [78]. Zhang et al. showed that currently-available static taint analysis tools produce many FPs and FNs in DroidBench apps that contain ICC- and lifecycle-related code similar to Figure 4.1. They also evaluated the tools with real-world apps and concluded that none of them was reliable. The increase of FPs increases the analysis cost and complicates the verification of analysis results. Also, increasing the complexity of analysis algorithms multiplies the analysis time, making it challenging to complete the analysis in a reasonable time.

4.1.4 Dynamic Taint Analysis

A dynamic taint analysis uses the target app's runtime semantics. For example, TaintDroid [54] performs taint tracking within the Dalvik virtual machine interpreter, and the Android models are not used. Since only the executed paths are analyzed, FPs, due to misestimating call flows and control flows, do not occur. The computation required for the

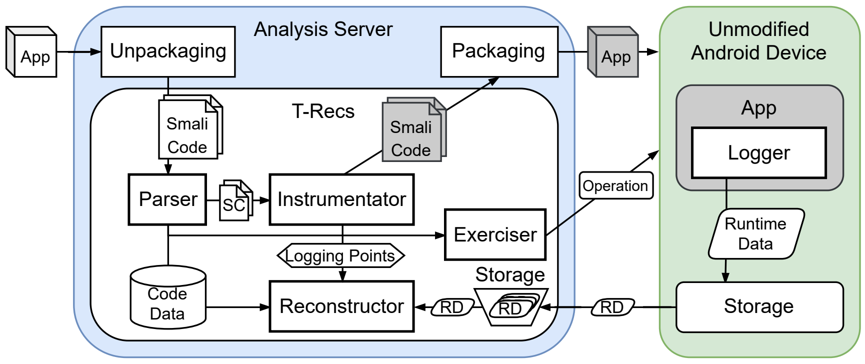


FIGURE 4.2: Overview of the approach.

Android modeling is no longer necessary, and the analysis time does not depend on the modeling.

However, Reaves et al. [80] discuss that TaintDroid is the most difficult to set up in comparison with static analysis tools they audited because TaintDroid requires the user to build the Android source code. Also, a supported device is not always available. Other current dynamic taint analyzers could be easier to set up; however, they have been barely examined in the community and are not effortlessly usable, which are discussed in Section 4.6.

TaintDroid's other drawback is that the app exercise needs to be executed every time running the taint analysis because the app exercise and the taint analysis are performed simultaneously in TaintDroid. Therefore, when the analyst changes the parameters of the taint analysis (e.g., the data to be tracked) and re-analyzes the same app, the app exercise also needs to be repeated, which incurs extra costs. It also distresses researchers who add new features to the taint analysis and evaluate them.

Another issue is the app exercise itself. Since a dynamic taint analysis only analyzes the executed part of the app's code, triggering the target behavior in the app is necessary. Monkey [109], a popular UI/application exerciser, exercises the app randomly. Random exercise is inefficient in triggering a leak in practice because sometimes a leak is only triggered by a particular sequence of UI operations. For example, some of the DroidBench apps require a specific sequence of operations to trigger leaks as shown by *callback1()* and *callback2()*, which must be called in this order to cause the leak (Figure 4.1). Finding such operation sequences using random input can take much time or fail to trigger leaks.

4.2 Approach

This section presents a new taint analysis system addressing the model accuracy, device dependency, and re-analysis cost issues.

4.2.1 Overview

The system is designed to be automatic for analyzing large-scale datasets. It performs a runtime-data-utilized taint analysis outside the Android framework to accomplish accuracy, usability, and small re-analysis costs. There are the following challenges:

- How to implement a mechanism to provide app runtime information outside the Android framework in a way that is effective for many real-world apps.

- What kind of app runtime information should be used to accurately reconstruct the app execution and track information flows outside the Android framework.
- Since the system requires the app's runtime data, how to automatically exercise the app to trigger the target behavior should be addressed.

The system consists of five components that address these challenges. Figure 4.2 shows the overview of the system. After the user plugs the unmodified Android device into the analysis server, the setup is finished, and the analysis server first unpackages the app and extracts the app's Smali code.

- **Parser** extracts the app's information from the Smali code to reduce the information to be logged.
- **Instrumentator** injects the logger into the app code. It also provides the logging point information to the reconstructor.
- **Logger** saves the app runtime information at the app's bytecode level. It is independent of specific Android devices and versions and requires no device modification, such as rooting. The logs are eventually stored in the analysis server.
- **Reconstructor** reproduces the app execution, including call-, control-, and dataflows on the analysis server based on the parsed and logged data. The logs are saved in the storage of the analysis server so that the reconstructor can be executed separately from the logger.
- **Exerciser** cooperates with the reconstructor to automatically trigger leaks caused by ICC, callbacks, and lifecycles in the DroidBench apps.

The rest of this section describes each component.

4.2.2 Parser

The parser extracts class, method, field, and instruction information from the app's Smali code. The parser maps each class and method (e.g., class *Leaker* and methods *init()*, *callback1()*, and *callback2()* in Figure 4.1) to distinguish between in-app and API methods, and the instrumentator uses the results. The parser also distinguishes fields (e.g., *imei* in Figure 4.1) implemented in subclasses and superclasses because the classes can have the same name fields, and the fields are not distinguishable based on their names. The parser also extracts fields' default values hard-coded in the app code.

4.2.3 Instrumentator

The instrumentator injects the logging code (i.e., the logger) into the target app's Smali code. In this section, first, logging points are described. Then, logging-method construction is explained. Next, the type-conflict problem and a solution are discussed. Lastly, DEX-related problems and solutions are described.

```
 1  .class LLeaker;
 2
 3  imei:Ljava/lang/String;  // field
 4
 5  .method public constructor <init>()V
 6          invoke-static {}, LTRecsLog;->Log_1_5()V
 7
 8          invoke-direct {p0}, Landroid/app/Application;-><init>()V
 9          invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_7_p0(Landroid/app/Application;)V
10  .end method
11
12  .method public callback1()V
13          invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_10_p0(LLeaker;)V
14
15          invoke-virtual {v0}, Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/String;  // source
16
17          move-result-object v1
18          invoke-static/range {v0 .. v0}, LTRecsLog;->Log_1_14_v0(Landroid/telephony/TelephonyManager;)V
19          invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_14_v1(Ljava/lang/String;)V
20
21          if-eqz v1, cond_0
22
23          iput-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field setter
24
25          :cond_0
26  .end method
27
28  .method public callback2()V
29          invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_23_p0(LLeaker;)V
30
31          iget-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field getter
32          invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_25_v1(Ljava/lang/String;)V
33
34          invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I  // sink
35          invoke-static/range {v0 .. v0}, LTRecsLog;->Log_1_27_v0(Ljava/lang/String;)V
36          invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_27_v1(Ljava/lang/String;)V
37  .end method
```

FIGURE 4.3: Example of the instrumentation applied to the code in Figure 4.1. The red-colored lines are logging points injected into the code.

Logging Points

Figure 4.3 shows an example of the instrumentation. The red-colored lines are logging points in the app code where the logging method invocations are injected. The logging methods are static and are called by the *invoke-static/range* instruction. An argument register is passed to the logging methods with brackets for logging the register's value. For example, *p0* is passed to the logging method by *{p0 .. p0}* in Line 9, and *p0*'s value will be logged. The subsequent *LTRecsLog;→Log_1_7_p0* specifies the called class and method names of the logger. *TRecsLog* is the class name and *Log_1_7_p0* is the method name. The method name consists of the target class identifier, the original instruction line number in the code before the instrumentation (Figure 4.1), and the register name, which are *1*, *7*, and *p0* respectively. The class identifier is assigned to each class (i.e., each Smali file) in the app. The following *(Landroid/app/Application;)* is the data type of the argument *p0*. *V* at the end is the data type of the method's return value and is void because the logging methods return nothing. The logging method in Line 6 has no argument, and no value is logged at the point. The logging points are as follows:

- immediately after field-getter instructions to save values loaded into the destination registers because fields can be modified outside the app code (e.g., Line 32 in Figure 4.3). Also, logging at static-field operators informs the reconstructor about the timing of the *clinit* invocation.

- right after *monitor-enter*, *const-class*, and *check-cast* instructions and catch labels.
- immediately after method calls (e.g., Lines 9, 18, 19, 35, and 36 in Figure 4.3) and at the head of each method in the app code (e.g., Lines 6, 13, and 29 in Figure 4.3). The logs are used to determine method call relationships accurately, which is explained in Section 4.2.5. Argument values are also recorded at the head of the method and used for argument mapping. The logger skips value logging for constructors because constructors have an uninitialized object reference as its base object at the method head (e.g., Line 6 in Figure 4.3). The return value is recorded for API method calls at the corresponding *move-result* (e.g., Line 19 in Figure 4.3) and used for the return value mapping, described in Section 4.2.5. The value of reference-data-type arguments is also recorded after the method call since the method may modify the arguments.

The instrumentator considers reducing the instrumentation code volume for app runtime performance. Results of arithmetic and logic operations and conditional branches (e.g., Line 21 in Figure 4.3) are not logged to reduce the amount of instrumentation code. Also, no logging code is injected to the end of each method (e.g., Lines 10, 26, and 37 in Figure 4.3). Instead, the reconstructor simulates the operations on the server based on reproduced register values.

Logging-Method Construction

Figure 4.3 indicates that a static logging method is constructed for every instrumented instruction. It avoids using local variables in the instrumented methods to reduce the impact on the original code.

Suppose all the logging points call the same logging method. In that case, each logging point must provide information, including the class identifier, line number, and register name to the logging method (i.e., each logging point needs a local variable to keep the information). However, introducing an additional local variable for the logging to a method can fracture the original code. A method can use registers *v0* to *v65535* for the method's local variables and parameters in Smali. The method's local variables are first assigned to registers from *v0*. Then, the method's parameters are assigned to registers. For example, if a method has 14 local variables and two parameters, the local variables use registers *v0* to *v13*, and the parameters use *v14* and *v15*. Assuming that an additional local variable is used for the logging, the local variables now use registers *v0* to *v14*, and the parameters use *v15* and *v16*. The second parameter's register is changed from *v15* to *v16*, which is not acceptable because whereas the first 16 registers *v0* to *v15* can be operated by all the instructions, *v16* and later registers are limited that only specific instructions can operate the registers. As a result, the instrumentator must rewrite the original code's instructions related to the second parameter, which is complex and better to be avoided. Therefore, the logging is designed to use no local variables in the instrumented methods. Since sharing a logging method among multiple logging points requires an additional local variable, a logging method is constructed for every logging point.

```
 1  :try_start_0
 2  invoke-static {}, LClass1;->method1()I;  // return an integer value
 3
 4  move-result v1                  // originally set the return value to v0
 5  invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_4_v0(I)V
 6  move v0, v1
 7
 8  int-to-float v0, v0                           // convert integer to float
 9  goto :goto_0                                  // jump to 14
10  :try_end_0
11  .catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0
12  :catch_0          // if an exception occurs from 1-10, jump to here
13  invoke-static/range {}, LTRecsLog;->Log_1_11()V
14  :goto_0
15  invoke-static {v0}, LClass2;->method2(F)V;     // v0 must be float
```

FIGURE 4.4: Instrumented code of exception handling. The modified part and injected code are red-colored.

```
FATAL EXCEPTION: main
  Process: com.sample, PID: 4022
  java.lang.VerifyError: Rejecting class com.sample.myclass
  register v0 has type int but expected float
```

FIGURE 4.5: An example of an error message indicating that the verifier detected a type conflict in register *v0*.

Type-Conflict Problem

The instrumentation code must not cause errors while the app is running. Solving the type-conflict problem discussed in [110] is challenging. Instrumenting an app can make data types of a register potentially conflicted within a method of the app.

Figure 4.4 shows an example of the instrumentation applied to an exception-handling code. The modified parts of the code are red-colored. A try block starts at *:try_start_0* in Line 1 and ends at *:try_end_0* at Line 10. If no exception occurs between Line 1 and 8, the *goto* instruction changes the program counter to *:goto_0* in Line 14, and *invoke-static* instruction in Line 15 is subsequently executed after the try block. In the original code, only *invoke-static* in Line 2 can cause an exception in the try block. Thus, the catch block would never be executed after the *invoke-static* in Line 2 is successfully executed, and the instructions in Lines 4, 8, and 9 are necessarily executed. In other words, in the original code, after an integer value is assigned to *v0* in Line 4, the value is certainly converted to float by *int-to-float* in Line 8.

On the other hand, if an exception occurs in the try block, the execution point is changed to *:catch_0* in Line 12. Then, instructions in Lines 13-15 are subsequently executed. In the code after the instrumentation, *invoke-static/range* in Line 5 can cause an exception in the try block, indicating that *int-to-float* in Line 8 can be skipped, and *v0* holds an integer value when the instructions after Line 12 are executed. In that case, the type-conflict problem occurs because *v0*'s value must be float for *invoke-static* in Line 15. In this way, the instrumentation can introduce a new exceptional flow leading to the type-conflict problem.

Since a verifier module of the Android runtime system always assumes that *invoke* causes exceptions, it detects the type conflict and terminates the app execution. The verifier generates an error message suggesting that a class was rejected because register *v0*'s type can be integer when the type must be float (Figure 4.5). To address the problem, Balachandran et al. developed a register-type separation technique, which rewrites the whole code of the app [110]. On the other hand, the approach replaces the destination register of *move-result* in a try block (e.g., *v0* at Line 4) with an unused register. The approach is called temporary-register technique in this paper. For example, *v1* is used as a temporary register instead of *v0* in Line 4, and the logger saves the *v1*'s value as the *v0*'s value in Line 5. After the logging, the value is moved from *v1* to *v0* in Line 6 to maintain the semantics. The technique is also applied to *move-exception*.

DEX-Related Problems

The instrumentator should avoid the 64K problem [111], which restricts a DEX file to containing 65,536 method references at most. After the logging methods are injected, the instrumentator counts the number of method references in Smali files in each DEX file, which is a directory when the app is unpackaged. Then, the instrumentator rearranges the Smali files in a DEX directory into multiple DEX directories if the DEX directory contains more method references than the maximum.

The instrumentator also detects long-distance jumps between an if-statement and its jump destination. When an if-statement uses a 16-bit address to specify the jump destination, the jump distance could exceed the limit because of the injected code, and the app repackaging would fail. The instrumentator detects such jumps and replaces the jumps with a *goto/32* statement that uses a 32-bit address.

4.2.4 Logger

The logger is injected into the app code by the instrumentator. Then, T-Recs repackages, installs, and launches the app on the Android device, and the logger is also executed. The logger targets primitive-data-type values, class object representations, string values of String type classes, array representations, and array elements' values.

The logger converts the class object representations into strings to write them into a log file. However, the formats of string representations depend on the class's *toString()* implementation. Also, the *toString()* should not be used when the class overrides the method because calling the method can affect the app's behavior (e.g., it could change a field value). Therefore, the logger explicitly invokes *getClass().getName()* and *hashCode()*. The logger uses a ring buffer to keep logs in the memory and reduce the number of writing to the disk for the app runtime performance with a large number of logging.

After the app is exercised for a time specified by the analysts, T-Recs terminates and uninstalls the app. Then, T-Recs collects the log file from the device and saves it in the analysis server.

4.2.5 Reconstructor

The reconstructor reproduces the app execution, consisting of call-, control-, and dataflows, based on the information obtained by the parser, the instrumentator, and the logger. Simultaneously, the reconstructor propagates taints to track information flows.

Figure 4.6 shows an example of reproducing the execution of the code in Figure 4.3. The log file is obtained by executing the code in Figure 4.3 and is stored in the storage of the analysis server so that the reconstructor can be performed independently of the logger. The log format is *PID:TID:ClassID_LineNumber*. PID and TID are the identifiers of the process and

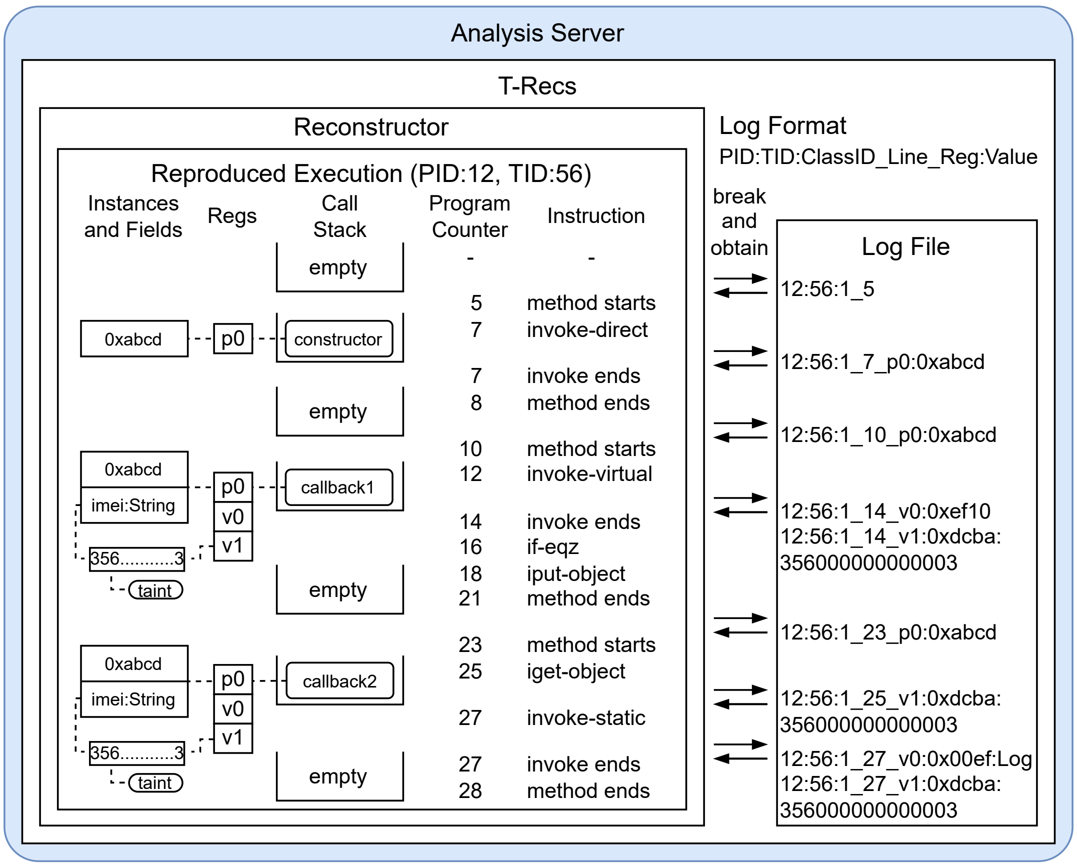


FIGURE 4.6: Example of app-execution reconstruction.

thread that executed the logging code. ClassID is the identifier of the class. LineNumber is the instrumented instruction's line number in the original code shown in Figure 4.1. The log is followed by *_RegisterName:RegisterValue* if the instrumented instruction has an operand register to be logged.

The reconstructor reproduces the execution with PID 12 and TID 56 in Figure 4.6. The reproduced execution includes currently-executed instruction, program counter, call stack, registers, instances, and fields. The call stack is empty at first, and the reconstructor starts by obtaining the first log *12:56:1_5*. The reconstructor sets the program counter 5 and simulates the instruction. Since program counter 5 is the head of *constructor*, the reconstructor creates its stack frame and pushes it to the call stack. Then, the reconstructor increments the program counter. The next instruction is *invoke-direct* at program counter 7, which is another logging point, and the reconstructor breaks the reproduction and obtains the next log *12:56:1_7_p0:0xabcd*. The reconstructor creates register *p0* and instance *0xabcd* based on the log. The reconstructor increments the program counter, reaches the end of the method, and removes the method's stack frame. Since the call stack is now empty, the reconstructor breaks the reproduction and obtains the next log, and the next method's reconstruction starts at program counter *10* in *callback1()*. By repeating these steps, the reconstructor reproduces the execution. The rest of this section explains how the reconstructor simulates register values, control flows, and call flows, and the taint propagation is described at the end.

Register Values

The reconstructor reproduces register values based on logged object identifiers, strings, and primitive-data-type values. It considers registers' data types: primitive-data-type, reference-data-type, and class references.

Primitive-data-type values are reproduced based on the logs and data extracted by the parser. Boolean values, true and false, are represented by numeric values, 1 and 0, respectively, for using the values with branches (e.g., *if-eqz*). Unary and binary operations, such as numerical and logical calculations, with primitive-data-type values, are simulated by the reconstructor. The reconstructor explicitly uses the same bit length to obtain the same calculation results as the actual execution.

Arrays and classes are reference-data-type, and registers of the data type hold object references. In the reconstructor, simulated registers hold references to array and class instances as same as the actual execution (e.g., register *p0* holds a reference to instance *0xabcd* in Figure 4.6). Null values are represented by the numeric value 0, which is compatible with branches (e.g., *if-eqz*). An array's elements are logged and used to simulate array operations. The reconstructor also supports multidimensional arrays. The reconstructor manages fields of classes and handles static fields as a global area. When the app accesses an uninitialized field, the reconstructor simulates default values, 0 for numeric values and null for objects.

When multiple threads write and read the same field simultaneously, the reconstructor detects the timing of each operation based on logs at the *monitor-enter*. Figure 4.7 shows an example of instrumented code with *monitor-enter* and *monitor-exit* instructions. These instructions enable an app to perform exclusive control to maintain consistency when multiple threads use the same data in, for example, a field. In this example, *thread1()* sets a value to field *Leaker.imei* at Line 7, and *thread2()* gets the value from the field at Line 19. Since these instructions are placed between *monitor-enter p0* and *monitor-exit p0*, they are executed one at a time. The reconstructor can detect their execution order based on the logs generated at Lines 5 and 17, and the data flow from *v1* in Line 7 to *v1* in Line 19 is accurately reproduced. On the other hand, if an app does not use exclusive control, the reconstructor cannot reproduce the execution order of instructions in multiple threads accurately. However, in such case, the impact of incorrect reproduction might be small because the app's developer also disregards the execution order.

Class references are generated by *const-class* instructions and used by branches and method calls. The reconstructor simulates class references based on logged object representations.

Control Flows

The reconstructor reproduces control flows in each method of the app, which consists of conditional branches (i.e., *if* and *switch*), unconditional jumps (i.e., *goto*), and exceptional flows (i.e., *try*, *catch*, and *throw*). App code is written in Dalvik executable (DEX) bytecode [112], which is register-based, and conditional branches operate on registers. Hence, the reconstructor simulates conditional branches based on the reproduced register values (e.g., program counter 16 in Figure 4.6).

Simulating exceptional flows requires the detection of exception sources and exceptional jump destinations. The reconstructor detects exception sources based on simulation results (e.g., ArrayOutOfBoundException by simulating arrays) and the logs (e.g., exception-causing calls by checking the completion of each call). The reconstructor checks a log that

```
.method public thread1()V
      invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_1_p0(LLeaker;)V

      monitor-enter p0
      invoke-static {}, LTRecsLog;->Log_1_3()V

      iput-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field setter

      monitor-exit p0
      return-void
.end method

.method public thread2()V
      invoke-static/range {p0 .. p0}, LTRecsLog;->Log_1_11_p0(LLeaker;)V

      monitor-enter p0
      invoke-static {}, LTRecsLog;->Log_1_13()V

      iget-object v1, p0, LLeaker;->imei:Ljava/lang/String;  // field getter
      invoke-static/range {v1 .. v1}, LTRecsLog;->Log_1_15_v1(Ljava/lang/String;)V

      monitor-exit p0
      return-void
.end method
```

FIGURE 4.7: Instrumented code with *monitor-enter* and *monitor-exit* instructions.

Application Space

```
class MainClass {
   main() {
      Leaker leaker = new Leaker();
      SystemClass1.method1(leaker);   }  }
```

Framework Space

```
class SystemClass1 {
   method1 (appClass) {
      appClass.callback1();   }   }
```

FIGURE 4.8: Simplified example of source code causing an implicit control flow transition.

must appear right after a finished call, and if the log is not found, the reconstructor understands that an exception is caused during the call. For example, the reconstructor detects that the call at Line 15 in Figure 4.3 causes no exception based on log *12:56:1_14_v0:0xef10* (Figure 4.6). In the same way, the reconstructor detects whether a *check-cast* instruction throws an exception. The reconstructor also breaks the trace and checks the next log at throw instructions. The reconstructor detects exceptional-jump destinations based on the logs from catch blocks (e.g., the logging point at Line 13 in Figure 4.4), which can be in a different method from the exception source.

Call Flows

There are various patterns of method calls involving callbacks, lifecycles, ICC, reflection, threading, and constructors. It indicates that only one-to-one mapping of parameters and arguments fails to detect dataflows from a caller to a callee. Also, the return value from a caller to a callee is not one-to-one because a callee can return a value to outside the app code, or a caller can receive a returned value from outside the app code.

The logs provide the reconstructor with the timings of the starting and ending of each method invocation and the timings of starting methods. The reconstructor breaks the trace when it reaches a method invocation instruction and checks the next log. If the following log is generated at the next line of the invocation, the invoked method is an API, not implemented in the app. If the following log is a method head's, an in-app caller-callee relationship is detected. The reconstructor utilizes the logged register values to match parameters and arguments from the caller to the callee. When the callee is finished, the reconstructor matches the return value from the callee to the caller based on the logged register values.

When the app executes multiple threads, all the threads' logs are mixed in the log file. The reconstructor distinguishes threads using each log's process and thread identifiers. Suppose the next log has new process and thread identifiers. In that case, the reconstructor considers that a new thread is starting and matches the base object's representation to the previously created instance's representations.

Detecting implicit control flow transitions facilitated by the callback mechanism in the Android framework is challenging for static taint analyzers [36]. Figure 4.8 shows an example of source code causing an implicit control flow transition. In the application space, *MainClass.main()* creates and passes a *Leaker* instance to *SystemClass1.method1()* in Lines 3 and 4. Then, in the framework space, *SystemClass1.method1()* invokes *appClass.callback1()* in Line 3, which is *Leaker.callback1()*, defined in Figure 4.1. *Leaker.callback1()* obtains IMEI, which can eventually be leaked. Therefore, a taint tracker must detect this control flow transition. The reconstructor cannot detect the relationship between *SystemClass1.method1()* and *Leaker.callback1()* because it occurs in the framework space. However, such a relationship is unnecessary because the reconstructor can detect the leak in Figure 4.1 based on logs generated at *Leaker.callback1()* and *Leaker.callback2()*. Also, based on PIDs and TIDs in the logs, the reconstructor can detect the exact time sequence of the method executions in a thread. On the other hand, if a callback method is executed in a different thread, the reconstructor identifies the time sequence as described in Section 4.2.5.

The reconstructor resolves the target class and method names of reflective calls to detect calls of taint sources and sinks. The reconstructor uses the argument values of the calls. Also, the reconstructor must consider that a taint source or sink can be called with an in-app class inheriting the class of the taint source or sink as the base object. The reconstructor resolves the called method's superclass based on class hierarchy information extracted by the parser.

There is a concern that the reconstructor may take a long time to analyze loops with a large number of iterations. In the preliminary investigation of the DroidBench apps, such loops were found in the method *computePi()* in PI1 from the category Emulator Detection (ED). Figure 4.9 shows *computePi()*'s source code. The method has no parameters, and the return value is not used by the caller (condition 1). In addition, the method has no API invocation or field operation in the method body (condition 2). The method does not affect anything outside the method, and the reconstructor can safely skip the method. Therefore, by checking the two conditions, the reconstructor automatically detects such a method as an anti-analysis technique and skips it.

Taint Propagation

Taint propagation is required at DEX bytecode instructions [112] and across API method calls to track information flows. Taint propagation rules for DEX bytecode instructions are well developed in previous studies such as TaintDroid [54], and the reconstructor utilizes

```
public static double computePi() {
    double n = 999999999;
    double sequenceFormula = 0;
    for (int counter = 1; counter < n; counter += 2) {
        sequenceFormula = sequenceFormula
            + ((1.0 / (2.0 * counter - 1))
                - (1.0 / (2.0 * counter + 1)));
    }
    double pi = 4 * sequenceFormula;
    return pi;
}

// Caller
public void theCaller() {
    computePi();
}
```

FIGURE 4.9: Source code of *computePi()* in case PI1 from category Emulator Detection.

the same rules. TaintDroid assigns taints to registers, but the reconstructor assigns taints to simulated class instances. For example, in Figure 4.6, the reconstructor detects the execution of the taint source at the program counter 14 and introduces the taint by assigning the taint mark to the string instance *356..3*, which will be stored in the field *imei* of the object *0xabcd*. When a reference to a class is moved between registers by register operations, the taint propagation is implicitly achieved, which is an advantage in simulating class instances. When a class instance field is operated through different registers holding the same reference (i.e., aliasing), the operations are implicitly applied to the same field of the same instance. For example, registers *p0* in *callback1()* and *p0* in *callback2()* reference the same object *0xabcd* with the field *imei* holding the tainted string in Figure 4.6.

Previous studies developed conservative rules [65] and automatic model generators (e.g., StubDroid [38]) for propagating taints across API method calls. The reconstructor, performing on the analysis server, can be equipped with current approaches used by static and dynamic taint trackers. In this study, an approach that conservatively propagates taints is simply used. There are some dataflows that the conservative rules cannot track. The reconstructor considers Intent, Message, Bundle, Shared Preferences, Parcel, and files. The reconstructor uses API class names and matches values between setter and getter methods. By propagating the taint status, the reconstructor not only assigns the taint but also removes the taint. It can refine over-tainting caused by the conservative rules and reduce FPs.

4.2.6 Exerciser

After the app is installed on the Android device, the exerciser performs app exercise operations to trigger leaks in the app. The exerciser focuses on how to exercise the DroidBench apps in this paper. Table 4.1 shows the necessary operations to trigger leaks in the DroidBench apps. It also shows commands to perform each operation and necessary information, such as the app's package name and activity name, which are passed to the commands. Leaks in some of the DroidBench apps can be triggered by only launching them. On the

TABLE 4.1: App exercise operations necessary to trigger leaks in the DroidBench apps.

Operation	Command	Necessary information
Launch app	monkey	app's package name
Kill app process	ps and kill	app's package name
Rotate screen	settings put system user_rotation	-
Press home button	input keyevent	-
Press back button	input keyevent	-
Start activity	am start	activity name
Send broadcast	am broadcast	receiver name
Start service	am start-service	service name
Tap screen	input tap	coordinates

other hand, some of the DroidBench apps require a sequence of operations specific to individual apps. Therefore, exercising the DroidBench apps using random input can take much time until a specific operation sequence is performed by chance.

The exerciser shortens the analysis time by triggering leaks in the DroidBench apps as quickly as possible. The exerciser iteratively runs the app and executes the reconstructor to detect information flows. The exerciser checks the reconstructor's result, and if the number of taint marks increases from the previous reconstruction result, the exerciser saves the current sequence of operations and uses it in the next turn.

Algorithm 2 shows the exerciser's pseudocode. The arguments are the app's data (*app*), a device with the app is installed (*device*), and the maximum number of operations to be performed (*max_op_num*). There are two loops in the procedure (Lines 7 and 11). In the outer loop, exercise operations that increase the taint marks (*taint_increasing_ops*) are performed in Lines 8-9, and performable operations (*performable_ops*) are obtained in Line 10. The performable operations include the operations shown in Table 4.1. For example, the coordinates of the app's UI buttons are detected, and the tap operation is prepared for each button. Then, in the inner loop, *taint_increasing_ops* are performed, and one of the performable operations is performed in Lines 15-19. The log is obtained in Line 20, and the reconstructor is executed with the log in Line 21. Based on the newly found leaks and the number of taint marks (*leaks* and *taint_num*), the exerciser stops the procedure (Lines 25-26), exits the inner loop (Line 27-33), or inserts *op* to *performable_ops*' head to retry the same operation (Line 34-35). Some DroidBench apps change their behavior depending on random numbers, and the number of taint marks can be different even for the same operation. Therefore, the exerciser retries the same operation as long as the number of taint marks decreases to trigger the app's information-flow-causing behavior. When the inner loop is finished, the outer loop also ends if the number of taint marks does not increase (Lines 38-39).

Also, determining when the exerciser exits is essential. The exerciser should not stop when a leak is detected, as some of the apps cause multiple leaks. The exerciser should also not run indefinitely, as reaching full coverage in runtime is very difficult. Therefore, in addition to the condition (Line 38-39), the exerciser stops when one of the following conditions is satisfied. First, the exerciser limits the number of performed operations (Line 12). Second, the exerciser exits when all the leaks are found in the app (Line 25). The *max_leak_num* is calculated before the exercise (Line 6) and is the number of all possible combinations of taint sources and sinks, including reflective calls, in the app extracted by the parser.

```
@Override
public void onLowMemory() {
    TelephonyManager telephonyManager =
        (TelephonyManager) getSystemService(
            Context.TELEPHONY_SERVICE);
    imei = telephonyManager.getDeviceId(); //source
}
```

FIGURE 4.10: Taint source called in *onLowMemory()* appeared in case RegisterGlobal2 from category Callbacks.

In addition, the exerciser lets the reconstructor simulate triggering non-triggerable callback methods. A callback method *onLowMemory()* is barely called in the DroidBench apps because the Android OS executes it only with memory-consuming apps. In the DroidBench test cases, five apps are identified to contain the callback method. The callback method executes a taint source, sink, or both. For example, case RegisterGlobal2 from category Callbacks overrides *onLowMemory()* to invoke a taint source (Figure 4.10). The exerciser tells the reconstructor to trigger *onLowMemory()* apart from the actual runtime. The reconstructor reproduces the app execution without the app's runtime information (i.e., without breaking the reproduced execution at logging points) and detects information flows and leaks caused by *onLowMemory()*.

4.3 Implementation

Python is used to implement all the components except the logger. The logger is implemented with the Smali language. The system is called T-Recs and is about 17,000 lines of code.

T-Recs uses the Apktool [113] version 2.6.1 to unpackage and repackage the apps. Apktool employs Baksmali [114], a disassembler for DEX bytecode, which converts DEX bytecode to mnemonic representation. The conversion generates text files with *.smali* extension. Apktool also has a DEX bytecode assembler, Smali [114], which converts *.smali* files into DEX bytecode format and creates an APK file. After the APK file is signed, it can be installed and executed on Android devices.

The reconstructor uses Python's references, exceptions, and lists to reproduce references, exceptions, and arrays in the app execution. NumPy is used to simulate the same bit length of numeric values in the reconstructor. The instrumentator performs the temporary-register technique, explained in Section 4.2.3, only for methods in that two more registers are available.

The logger currently targets a limited depth of arrays, which is two-dimensional. The supported level of depth can be trivially expanded by modifying the logger to record more items in multidimensional arrays. However, the modification could affect the app runtime performance.

As Section 4.2.6 explains, the exerciser is a prototype only for the DroidBench apps. The callback-method triggerer is implemented to reproduce the execution of *onLowMemory()*, the only method that cannot be triggered on Android devices in the DroidBench apps. Since taint sources must be executed before taint sinks to cause the leaks, the timings of the method

Algorithm 2 App exercise procedure

```
procedure EXERCISE($app, device, max\_op\_num$)
    $op\_num \leftarrow 0$
    $prev\_taint\_num \leftarrow 0$
    $found\_leaks \leftarrow$ empty list
    $taint\_increasing\_ops \leftarrow$ empty list
    $max\_leak\_num \leftarrow$ get_max_leak_num($app$)
    while true do
        stop $app$ and remove the logs on $device$
        perform $taint\_increasing\_ops$
        generate $performable\_ops$
        while $performable\_ops$ is not empty do
            if $op\_num > max\_op\_num$ then
                return
            end if
            stop $app$ and remove the logs on $device$
            perform $taint\_increasing\_ops$
            $op \leftarrow$ pop an item from $performable\_ops$
            perform $op$
            $op\_num \leftarrow op\_num + 1$
            $log \leftarrow$ logs obtained from $device$
            $leaks, taint\_num \leftarrow$ reconstructor($log$)
            if $leaks$ not in $found\_leaks$ then
                $found\_leaks \leftarrow leaks | found\_leaks$
                $leak\_num \leftarrow$ length of $found\_leaks$
                if $leak\_num = max\_leak\_num$ then
                    return
                else
                    $taint\_increasing\_ops \leftarrow$ empty list
                    break
                end if
            else if $taint\_num > prev\_taint\_num$ then
                append $op$ to $taint\_increasing\_ops$
                break
            else if $taint\_num < prev\_taint\_num$ then
                append $op$ to $performable\_ops$' head
            end if
        end while
        if $taint\_num \leq prev\_taint\_num$ then
            break
        end if
        $prev\_taint\_num \leftarrow taint\_num$
    end while
end procedure
```

execution are at each constructor's end and the whole reconstruction's end, which were determined based on the investigation of the DroidBench apps.

4.4 Evaluation

This section presents T-Recs' evaluation with a test suite and real-world apps. First, this section explains datasets. Then, it describes the compared tools and analysis results of each dataset. Lastly, it explains ethical considerations.

4.4.1 Datasets

The following datasets were used.

DroidBench 3.0

DroidBench is a popular test suite initially published in 2014, covering a wide range of language- and Android-specific categories. DroidBench had 19 categories and 190 test cases in total when this paper was written. This paper focuses on 158 test cases in 13 categories supported by current static taint trackers: FlowDroid, Amandroid, and DroidSafe [78, 115] to evaluate how T-Recs outperforms the trackers in accuracy with the supported cases. The categories are Aliasing (A), Android Specific (AS), Arrays and Lists (AL), Callbacks (C), ED, Field and Object Sensitivity (FO), General Java (GJ), ICC, Lifecycle (L), Reflection (R), Reflection ICC (RICC), Threading (T), and Unreachable Code (UC).

Popular apps from the Google Play Store in 2016

Analysis accuracy, time, and success rate were evaluated for detecting privacy leaks in real-world apps. Since TaintDroid detects leaks of sensitive information, such as IMEI and IMSI, a dataset in this evaluation must contain many apps that obtain and leak the information. Popular apps in the Agrigento dataset were collected from the Google Play Store and have such suspicious apps: 22 apps leaking IMEI and six apps leaking IMSI [21]. They were collected in June 2016. The app set contains 96 apps given by the authors of [21]. A complete list of hashes of the apps mentioned in this section is given in Appendix A.

Varied dataset from the Google Play Store and Anzhi

The app set contains randomly-collected 19,943 apps from the Google Play Store and 19,537 apps from Anzhi, 39,480 apps in total, via AndroZoo [116] in September 2021 for evaluating the success rates of the compared tools' essential phases. Anzhi was selected as the representative of third-party app markets because Anzhi was the market with the largest app collection after the Google Play Store [116]. The apps from the Google Play Store support SDK versions from one to 28 (i.e., 16 codenames), and the Anzhi apps support SDK versions from one to 25 (i.e., 14 codenames). The datasets vary in supported SDK versions. Also, the distribution of apps' supported SDK versions differs between the markets, and the tools were evaluated with a wide range of SDK versions.

Popular apps from the Google Play Store in 2021

The leak detection number and analysis time were evaluated with newer real-world apps. The app set contains 158 apps that appeared in the top chart list of free apps in the Google Play Store in July 2021. Since these apps are recently developed, 98% of them have *androidx.** packages [117]. Leaks caused by the packages were ignored because the packages are official and can be considered benign. Appendix A shows a complete list of hashes of the 158 apps.

4.4.2 Privacy Leak Detection in DroidBench 3.0

This section describes the evaluation results of DroidBench 3.0 to show T-Recs' superiority over current trackers in detection accuracy. It also discusses the analysis time.

TABLE 4.2: Tools compared to T-Recs in the evaluation with DroidBench 3.0.

Tool	Version
FlowDroid	2.9.0
FlowDroid$_{IC3}$	FlowDroid w/ IC3 0.2.0
Amandroid	3.2.1
DroidSafe	2016-Jun-23
DroidRA$_F$	2017-Apr-10 w/ FlowDroid
DroidRA$_A$	2017-Apr-10 w/ Amandroid
DroidRA$_D$	2017-Apr-10 w/ DroidSafe
IccTA	2016-Feb-21
RAICC	1.0
TaintDroid	4.3_r1
IntelliDroid	2018-Jun-20

Compared Tools and Setup

Table 4.2 shows the tools compared to T-Recs. Available static taint analysis tools were selected based on the study by Zhang et al. [78]: FlowDroid [118], FlowDroid$_{IC3}$ [118], Amandroid [119], DroidSafe [120], and DroidRA [121]. IC3 is obtained from the authors of [78]. In accordance with [78], DroidRA is used with FlowDroid, Amandroid, and DroidSafe, denoted by DroidRA$_F$, DroidRA$_A$, and DroidRA$_D$ respectively. The same tool options and taint source and sink definitions as the study [78] were used. Also, the comparison includes tools targeting ICC: IccTA [122] and RAICC [34, 123]. In addition, it includes TaintDroid [124] and IntelliDroid [125], which leverage a dynamic taint analysis.

Note that whereas the idea of the recording and reconstruction was initially realized in VTDroid [126], VTDroid was omitted from the evaluation. The decision is because VTDroid is specialized for specific flows, e.g., control dependencies and timing channels, which are not supported by the selected tools, including T-Recs.

The execution environment for T-Recs and the static analyzers is a ten-core (20 threads) 3.7GHz CPU and 128GB RAM. Devices of Zenfone 4 (Android 8.0.0) and Nexus 5 (Android 5.0.1) and an emulator of Nexus 9 (Android 8.0.0) were used for T-Recs to exemplify T-Recs' independency from specific Android devices and versions.

This section also involves TaintDroid with Nexus 4 (Android 4.3), the most popular and stable dynamic taint tracker for Android apps. In order to evaluate TaintDroid with DroidBench, TaintDroid was modified to support the taint sinks, which the original version of TaintDroid does not support. TaintDroid only supports HTTP/HTTPS transmission as the

TABLE 4.3: Taint sinks and corresponding modified files' paths.

Sink	Path
android.telephony.SmsManager: void sendTextMessage(java.lang.String, java.lang.String, java.lang.String, android.app.PendingIntent, android.app.PendingIntent)	frameworks/opt/telephony/src/java/android/telephony/SmsManager.java
android.util.Log: int i(java.lang.String, java.lang.String) android.util.Log: int e(java.lang.String, java.lang.String) android.util.Log: int v(java.lang.String, java.lang.String) android.util.Log: int d(java.lang.String, java.lang.String)	frameworks/base/core/java/android/util/Log.java
java.lang.ProcessBuilder: java.lang.Process start()	libcore/luni/src/main/java/java/lang/ProcessBuilder.java
android.app.Activity: void startActivityForResult(android.content.Intent, int) android.app.Activity: void startActivity(android.content.Intent) android.app.Activity: void setResult(Iandroid.content.Intent)	frameworks/base/core/java/android/app/Activity.java
java.net.URL: java.net.URLConnection openConnection()	libcore/luni/src/main/java/java/net/URL.java
android.content.ContextWrapper: void sendBroadcast(android.content.Intent)	frameworks/base/core/java/android/content/ContextWrapper.java

TABLE 4.4: Results of DroidBench. The second column shows the expected #leaks. Gray cells highlight accurate results.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
A (4)	1	1	0	0	1	1	0	1	1	0	1	1	0	0	0	1	1	1	0	1	1	0	0	0	1	1	1	0	1	0	0	1	0	0
AS (11)	8	8	0	0	7	1	1	7	1	1	4	0	4	7	1	1	7	1	1	4	0	4	7	1	1	6	0	2	5	0	3	5	0	3
AL (10)	4	4	0	0	4	3	0	4	5	0	1	4	3	4	4	0	4	5	0	1	4	3	4	4	0	3	5	1	1	0	3	1	0	3
C (15)	18	18	0	0	15	2	3	15	2	3	2	1	16	18	4	0	15	2	3	2	1	16	18	4	0	16	2	2	1	0	17	1	0	17
ED (15)	16	16	0	0	16	0	0	16	0	0	15	0	1	11	0	5	16	0	0	15	0	1	11	0	5	16	0	0	9	0	7	9	0	7
FO (7)	2	2	0	0	2	0	0	2	0	0	2	0	0	2	2	0	2	0	0	2	0	0	2	2	0	2	0	0	2	0	0	2	0	0
GJ (25)	22	22	0	0	18	4	4	18	4	4	5	2	17	22	2	0	18	4	4	5	2	17	22	2	0	18	5	4	10	0	12	12	0	10
ICC (18)	27	27	0	0	18	0	9	14	0	13	23	9	4	21	2	6	14	0	13	23	9	4	21	2	6	19	1	8	18	0	9	20	0	7
L (24)	21	21	0	0	14	1	7	14	1	7	6	2	15	21	9	0	13	1	8	6	2	15	21	9	0	16	1	5	9	0	12	9	0	12
R (9)	9	9	0	0	8	0	1	8	0	1	1	0	8	4	0	5	8	0	1	6	0	3	6	0	3	1	0	8	9	0	0	9	0	0
RICC (10)	21	21	0	0	2	0	19	2	0	19	4	0	17	5	0	16	2	0	19	4	0	17	5	0	16	2	0	19	19	0	2	19	0	2
T (6)	6	6	0	0	5	0	1	5	0	1	1	0	5	4	1	2	5	0	1	1	0	5	4	1	2	3	0	3	6	0	0	6	0	0
UC (4)	0	0	0	0	0	3	0	0	3	0	0	4	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	0	0	0	0	0
Sum (158)	155	155	0	0	110	15	45	106	17	49	65	23	90	119	28	36	105	17	50	70	22	85	121	28	34	103	18	52	90	0	65	94	0	61

taint sinks by default. Six files in the source code of TaintDroid were modified to support taint sinks in the DroidBench apps. Table 4.3 shows the added taint sinks and corresponding file paths. For more details, the modified code is available upon reasonable request.

Since TaintDroid does not have an app-exercise ability, a publicly-available hybrid analysis tool called IntelliDroid was used in combination with TaintDroid. IntelliDroid performs targeted execution and officially supports TaintDroid.

The exerciser was employed to exercise the apps for T-Recs automatically. As Section 4.2.6 explains, the exerciser requires a parameter that specifies the maximum number of exercise operations (Table 4.1) to be performed for an app on a device. The parameter value was determined based on a preliminary investigation. Each app was tested, and the number of exercise operations needed to trigger leaks in each app was obtained. The maximum was 27 for a case in the ICC category. Therefore, 30 was used as the maximum number of exercise operations in this evaluation.

Detection Accuracy

Table 4.4 shows the result. The expected leak numbers are obtained from [115]. The result shows that only T-Recs is 100% accurate. The parser, the instrumentator, and the logger successfully processed all the apps, the exerciser automatically triggered all the leaks, and the reconstructor successfully detected all the leaks.

Notably, in ED, the instrumentator did not inject the logger into the *computePi()* method in an app called PI1 and successfully kept the method execution time within the threshold. In addition, the reconstructor detected *computePi()* as a method that does not affect the execution and skipped it, resulting in successful leak detection.

T-Recs' independence of the analysis environment helps T-Recs analyze apps in ED. One of the apps, for example, triggers a leak only when specific files exist on the device. Prior to the evaluation, Nexus 4 (Android 4.3), Nexus 5 (Android 5.0.1), Zenfone 3 (Android 6.0.1), Zenfone 4 (Android 8.0.0), Pixel 4 (Android 10), and Pixel 6 (Android 12) were investigated.

TABLE 4.5: Analysis time for the DroidBench apps.

Tool	**Time**
T-Recs	4 hours 58 minutes
FlowDroid$_{IC3}$	14 minutes
FlowDroid	4 minutes
Amandroid	47 minutes
DroidSafe	15 hours 37 minutes
DroidRA$_F$	18 hours 56 minutes
DroidRA$_A$	19 hours 56 minutes
DroidRA$_D$	33 hours 52 minutes
IccTA	1 hour 26 minutes
TaintDroid	36 minutes
IntelliDroid	59 minutes

Among them, only Nexus 4 and 5 can trigger the leaks. Nexus 5 was included in this evaluation, and T-Recs successfully ran on the device and detected the leaks.

FlowDroid$_{IC3}$ and FlowDroid generate 15 and 17 FPs, respectively, in call-flow-related cases (AS, C, GJ, and L), control-flow-related cases (UC), and other cases (A and AL). UC is related to path sensitivity, which FlowDroid cannot consider. The dynamic taint trackers outperform the static taint trackers in this category. FlowDroid$_{IC3}$ and FlowDroid also produce some FNs. In particular, the tools produce nine and 13 FNs because of failure in intent tracking in ICC and 19 FNs in RICC because of failure to resolve reflective calls. The other static taint analysis tools, Amandroid, DroidSafe DroidRA$_F$, DroidRA$_A$, DroidRA$_D$, and IccTA, also produce a certain amount of FPs and FNs.

TaintDroid generates no FP, indicating that the tool is accurate. However, TaintDroid misses 65 leaks. IntelliDroid improves TaintDroid's result in three apps (four leaks) in GJ and ICC. Since some callbacks were successfully triggered, IntelliDroid should be adequate for more apps. The small number of improvements may be due to the quality of the tool, and increasing the quality may improve the result.

RAICC instruments none of the 158 apps. This is because the apps do not contain code targeted by RAICC. Since no leak detection is performed, the RAICC's result is excluded from Table 4.4. Section 4.6 discusses RAICC further.

Analysis Time

Table 4.5 shows the result. Each tool analyzed the apps one at a time. T-Recs' parser and instrumentator took 15 minutes, and the exerciser with the reconstructor took four hours and 43 minutes. T-Recs is the fifth slowest, but it is acceptable because it finishes within a reasonable time (one minute and 53 seconds per app). FlowDroid is the fastest, but the result would be different in real-world-app analysis because the benchmark apps have minimal code, and the static analysis time depends on the amount of code. On the other hand, T-Recs, TaintDroid, and IntelliDroid require an app execution time regardless of benchmarks or real-world apps. A result of real-world-app analysis is discussed in Section 4.4.3.

The analysis time of T-Recs can be shortened by improving the exerciser. The maximum number of exercise operations slightly influences the number of leaks detected and analysis time. If the maximum number of exercise operations is 40, T-Recs detects all the leaks and takes five hours and 13 minutes. It is 10% longer than the analysis with 30 as the maximum number of exercise operations. If the maximum number of exercise operations is 20, T-Recs fails to detect one leak in a case in the ICC category and takes four hours and 48 minutes. The analysis time is almost the same as one with 30 as the maximum number of exercise operations. Each analysis time is the average of three executions. On the other hand, T-Recs took 52 minutes in total with an ideal exerciser, which was manually created and contained a minimum set of operations to trigger all the leaks.

4.4.3 Privacy Leak Detection in Popular Apps 2016

This section compares T-Recs, FlowDroid, FlowDroid$_{IC3}$, Amandroid, DroidSafe, DroidRA$_F$, DroidRA$_A$, DroidRA$_D$, IccTA, TaintDroid, and IntelliDroid based on accuracy, time, and success rate for detecting privacy leaks in real-world apps. The tracking targets are those that the tools support: hardware identifiers (IMEI, IMSI, and ICCID), phone numbers, and location data.

Compared Tools and Setup

T-Recs uses the Pixel 3 with Android 9 and the computer explained in Section 4.4.2. Android 9 is the last version in which the hardware identifiers are accessible. For T-Recs and TaintDroid, each app is installed and launched on the Android devices, and each system waits for approximately 60 seconds and then uninstalls it. It was decided that the apps were not exercised, based on the results of the preliminary experiment, indicating that apps in the dataset cause leaks by simple operations, such as starting an app. A one-hour timeout is used per app for each of T-Recs and the static analyzers.

Having flawless taint sink definitions for T-Recs and the static taint analyzers is challenging because there are numerous candidates, which are API methods that may cause leaks. Since taint sink definitions must be prepared to detect privacy leaks, they were created in an ad hoc manner. API methods of network-related libraries were chosen. Also, API methods that write data to transmit it to the network were selected. These sink definitions were used for T-Recs and the static taint analyzers, and it was believed that none of the tools is particularly advantageous to producing more TPs. However, the taint sink definitions cannot be used universally. Also, taint sink definitions vary depending on what code the analysts attempt to find and should be prepared by the analysts on their own. Therefore, it should be clear that this paper does not offer taint sink definitions (i.e., out-of-scope). On the other hand, TaintDroid's default sink definitions were used for TaintDroid and IntelliDroid.

Detection Accuracy

Whereas establishing a ground truth is infeasible, correct leaks were obtained by manually searching the network dump for plaintexts (e.g., IMEI value 356000000000003 in Figure 4.6) and names (e.g., IMEI) of the target information. Also, transformed data (e.g., XXxxxxx==) reported by T-Recs and TaintDroid were searched. In the dynamic analysis, detecting no leak is correct if no leak occurred, and detecting a leak is correct if the leak occurred. Therefore, considering only leaks occurring on both T-Recs' and TaintDroid's devices is reasonable. Each alert of T-Recs and TaintDroid was compared with the network dump to verify that the leak occurred. If an alert includes transformed data, the app code was manually analyzed to confirm that the data contain the target information. The number of unique URLs in TaintDroid's alerts and the number of unique sink code locations in T-Recs' alerts were counted.

TABLE 4.6: Leak detection result. E indicates expected #leaks. ✗ indicates that the tool failed, and $✗_{IC3}$ indicates that IC3 failed.

#App	E	T-Recs			TaintDroid			IntelliDroid			FlowDroid		FlowDroid$_{IC3}$		Amandroid		DroidSafe		DroidRA$_F$		DroidRA$_A$		DroidRA$_D$		IccTA	
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	unsure	TP	unsure	TP	unsure	TP	unsure	TP	unsure	TP	unsure	TP	unsure	TP	unsure
1	10	6	12	4	4	8	6		✗		✗		$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	8	5	3	3	4	0	4	0	0	8	6	2	✗		✗		✗		✗		✗		✗		✗	
1	5	5	4	0	0	0	5	0	0	5	0	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	5	5	1	0	1	0	4	1	0	4	0	3	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	4	2	0	2	0	0	4	0	0	4	2	16	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	4	3	2	1	3	0	1	2	0	2	2	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	4	2	5	2	1	0	3	0	0	4	0	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	4	4	13	0	3	0	1		✗		4	2	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	3	3	0	0	1	0	2		✗		3	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	3	3	9	0	3	0	0	0	0	3	1	4	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	2	0	0	2	2	0	0	0	0	2	1	0	$✗_{IC3}$		✗		✗		1	0	✗		✗		✗	
1	1	1	2	0	1	1	0	0	0	1	✗		$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	1	0	0	1	1	0	0	0	0	1	1	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	1	1	4	0	1	1	0	0	0	1	0	1	✗		✗		✗		✗		✗		✗		✗	
1	1	0	0	1	0	0	1	0	0	1	0	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	1	1	0	0	1	3	0	0	0	1	0	1	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	1	1	0	0	1	0	0		✗		0	0	0	0	✗		✗		✗		✗		✗		✗	
1	1	1	2	0	0	0	1		✗		0	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0		✗		0	3	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	1	$✗_{IC3}$		✗		✗		0	1	✗		✗		✗	
1	0		✗		0	0	0	0	0	0	0	0	$✗_{IC3}$		0	0	✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0		✗		0	0	✗		0	0	✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	3	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	1	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	✗		✗		0	1	✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗		0	0
1	0	0	0	0	0	0	0		✗		0	0	0	0	✗		✗		✗		✗		✗		0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		0	0	✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	2	0	0	✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0		✗		0	1	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0		✗			✗		0	0	0	0	✗		✗		✗		✗		✗		✗	
1	0		✗		0	0	0	0	0	0	0	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		0	0
1	0	0	0	0	0	0	0	0	0	0	0	4	✗		✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗		✗	
1	0	0	0	0	0	0	0		✗		0	2	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗	
2	0	0	0	0	0	0	0	0	0	0	0	2	0	0	✗		✗		✗		✗		✗		0	2
2	0	0	0	0	0	0	0		✗		0	3	0	0	✗		✗		✗		✗		✗		✗	
2	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗		✗		✗		✗	
3	0	0	0	0	0	0	0	0	0	0	✗		✗		0	0	✗		✗		✗		✗		0	0
3	0	0	0	0		✗			✗		0	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
4	0	0	0	0	0	0	0	0	0	0	0	0	$✗_{IC3}$		0	0	✗		✗		✗		✗		✗	
4	0	0	0	0	0	0	0	0	0	0	✗		$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
8	0	0	0	0	0	0	0		✗		0	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	✗		✗		✗		✗		✗		✗	
16	0	0	0	0	0	0	0	0	0	0	0	0	$✗_{IC3}$		✗		✗		✗		✗		✗		✗	
96	59	43	57	16	27	14	32	3	0	56	20	58	0	0	0	0	0	0	1	2	0	0	0	0	0	4

Table 4.6 shows the result. The expected #leaks indicates the number of unique URLs with that the sensitive information leaked. T-Recs does not generate FP for apps without leaks. Since the analyst needs to check only the 18 apps with leaks, the impact of the FPs is small, suggesting that T-Recs is highly accurate. T-Recs has more FPs than TaintDroid because T-Recs' sink definition differs from TaintDroid. The conservative rules for API method calls explained in Section 4.2.5 may also be a factor. For the same reason, T-Recs generates more TPs than TaintDroid.

IntelliDroid only detects the three expected leaks in two apps, which are also detected by TaintDroid (Table 4.6). On the other hand, IntelliDroid misses many leaks that TaintDroid detects, demonstrating that introducing IntelliDroid into TaintDroid makes TaintDroid overlooks more leaks. Note that IntelliDroid finds a leak that does not occur in the environments of T-Recs and TaintDroid, showing its superiority over T-Recs and TaintDroid. However, this paper ignores the leak because improving code coverage of real-world apps is out-of-scope of this paper, as Section 4.5 also discusses.

FlowDroid's alerts were also verified based on T-Recs' results because the two tools use the same sink definitions. In addition, taint sources suggested by each alert were compared with taint sources identified in the network dumps, and 20 TPs were identified. On the other hand, as indicated by *unsure* in Table 4.6, the 58 alerts do not match with the network dumps. They are considered to be unsure leaks that could be either TP or FP because there were no resources for further high-cost verification. It was confirmed that T-Recs does not detect the unsure leaks because of the code coverage. All the unsure leaks are caused by codes outside the T-Recs' code coverage. The maximum, minimum, average, and median of the T-Recs' code coverages were 40.7%, 0.3%, 6.6%, and 4.3%, respectively. Note that 15 apps with zero code coverage were omitted from this calculation. In contrast, FlowDroid can analyze the entire code of each app, which is an advantage of static analysis. There is a trade-off between the coverage and accuracy, which is discussed in Section 4.5.

FlowDroid$_{IC3}$, Amandroid, DroidSafe, DroidRA$_A$, and DroidRA$_D$ detect no leaks. Note that the default definitions of taint sources and sinks are used for DroidSafe because changing the definitions requires modification of the source code of DroidSafe. Since DroidSafe with the default definitions fails to analyze all the apps, it would detect no leaks even if different source and sink definitions, such as the ones used by T-Recs and the other static analyzers, were used. Also, the developer clearly states that DroidSafe is unsuitable for analyzing apps published on the Google Play Store [127]. DroidRA$_F$ finds one TP and two unsure leaks, which are also detected by FlowDroid. IccTA detects no TP and four unsure leaks. These seven tools are not very effective in analyzing real-world apps.

On the whole, FlowDroid misses many leaks that T-Recs and TaintDroid detect, and FlowDroid's recall is low. At the same time, FlowDroid generates 58 unsure alerts, suggesting high verification costs. Therefore, T-Recs and TaintDroid are more practical than FlowDroid for privacy leak detection. Also, the other tools generate almost no alerts and cannot be used for privacy leak detection dependably.

Tracking Ability for ICC- and Reflection-Related Flows

Static analysis can usually detect more leaks with higher FP rates than dynamic analysis. However, the result shows that the dynamic analyzers (i.e., T-Recs and TaintDroid) detect more leaks than the static analyzer (i.e., FlowDroid). T-Recs detects 43 TPs; TaintDroid, 27 TPs; and FlowDroid, 20 TPs. One of the possible reasons is that FlowDroid failed to complete

TABLE 4.7: ICC- and reflection-related flows selected based on the DroidBench apps with that FlowDroid generates FNs.

Type	Description
1	*Activity.startActivity()* with a tainted argument
2	*Messenger.send()* with a tainted argument
3	Reflective call with a tainted argument
4	Reflective call with the tainted return value
5	Reflective call of a taint source

TABLE 4.8: #apps and #codes in parentheses in which the five code types are found and whether T-Recs and FlowDroid detect the leaks caused by the five code types. The row "any" gives #apps and #codes in which at least one code type is found.

	T-Recs			FlowDroid		
Type	flow	TP	FP	TP	FN	incompleted
1	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
2	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
3	4 (8)	2 (5)	0 (6)	0 (0)	1 (3)	1 (2)
4	4 (8)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
5	7 (7)	5 (15)	0 (11)	0 (0)	4 (9)	1 (6)
any	10 (20)	6 (18)	0 (13)	0 (0)	5 (12)	1 (6)

the analysis of apps in that the dynamic analyzers detect TPs. FlowDroid failed the analysis for two apps with one or more expected leaks (Table 4.6).

Another possible reason is the difference in information-flow-tracking abilities between FlowDroid and the other tools. Since FlowDroid mostly misses leaks in the DroidBench apps of ICC and RICC, this section is focusing on the ICC and RICC cases. Selected code types are shown in Table 4.7 based on the DroidBench apps with which FlowDroid generates FNs (Table 4.4). T-Recs' reconstructor was modified to identify and count the occurrences of the five code types. The experiment was conducted once the overall analysis was completed (i.e., after the results in Table 4.6 were obtained). Since the logs obtained by the logger were kept, only re-executing the reconstructor was needed. In other words, re-exercising the app is unnecessary when testing a new feature in the reconstructor, which is further discussed in Section 4.5.

Table 4.8 shows the number of apps and code points where the five code types are found by T-Recs (second column *flow*). It also shows the number of apps and code points where T-Recs detects leaks caused by the five code types (third and fourth columns). It also shows whether FlowDroid detects the TP leaks detected by T-Recs (fifth, sixth, and seventh columns). The result shows that four out of the five code types are found, and two of them cause leaks. Whereas six leaks of the third type and 11 of the fifth type are FP (i.e., the leaks are falsely detected), all the leak-detected apps are TP (i.e., no app is falsely detected by T-Recs). In contrast, FlowDroid fails to detect all of them. FlowDroid misses some of the leaks and fails to complete the analysis of some apps, as indicated by *incompleted*. Note that Table 4.8 excludes the result that FlowDroid detects none of the FP leaks detected by T-Recs. The results highlight the importance of tracking ICC- and reflection-related flows as the flows appear in the real-world apps as well as the DroidBench apps.

There can be other types of code with which FlowDroid generates FNs in the DroidBench apps. However, identifying the exact instructions preventing FlowDroid from tracking flows

TABLE 4.9: Analysis time for the privacy leak detection.

Tool	Time
T-Recs	3 hours 19 minutes
TaintDroid	2 hours 19 minutes
IntelliDroid	70 hours 6 minutes
FlowDroid	2 hours 34 minutes
FlowDroid$_{IC3}$	15 hours 33 minutes
Amandroid	82 hours 5 minutes
DroidSafe	20 hours 29 minutes
DroidRA$_F$	91 hours 32 minutes
DroidRA$_A$	95 hours 16 minutes
DroidRA$_D$	94 hours 55 minutes
IccTA	55 hours 39 minutes

requires debugging FlowDroid, which is unfamiliar to the author. Therefore, only ICC and reflection are focused in this section, and it is considered that the choice to be sufficient to show how T-Recs detects leaks that FlowDroid misses.

Analysis Time

Table 4.9 shows the result. TaintDroid is the fastest, FlowDroid is second, and T-Recs is third. Their results are not largely different. They did not reach the timeout in analyzing any apps. In comparison, FlowDroid$_{IC3}$ and the other tools took over 15 hours each. They failed many apps because of the timeout and are not suitable for the analysis of a set of real-world apps. For example, IC3 improves the performance of FlowDroid's ICC handling, but the result shows that IC3 cannot be finished in a reasonable time.

T-Recs' parser and instrumentator took 27 minutes; the app exercise, 138 minutes; and the reconstructor, 34 minutes. After the parser and the instrumentator process an app, the analyst can analyze the app to detect various information flows by reconfiguring and running the reconstructor and, if necessary, exercising the app on a device to acquire more code coverage. In other words, the parser and the instrumentator only need to be executed once for each app. Hence, reducing the analysis time for the parser and the instrumentator is a low priority. The two components have not been optimized in the current implementation by, for example, processing Smali files in parallel. Therefore, The two components were executed in 12 threads without conducting a further performance evaluation of them.

On the other hand, the reconstructor must be executed every time the analyst changes the configuration (e.g., taint source and sink definitions). Also, if the analyst needs more code coverage, re-trying the app exercise is necessary. Therefore, the rest of this section first discusses the analysis time taken by the reconstructor (Section 4.4.3), and then the analysis time for the app exercise is explained (Section 4.4.3).

T-Recs' and FlowDroid's Parallel App Analysis Time.

The reconstructor's time depends on the number of apps analyzed in parallel, which is determined by the available RAM size of the computer used. A computer with 128GB RAM was used, and each tool's memory was fixed to 120GB by using a Docker container limited to 120GB RAM with disabling the swap. Since the computer's CPU has ten cores (20 threads), the reconstructor was executed by changing the number of apps analyzed in parallel from

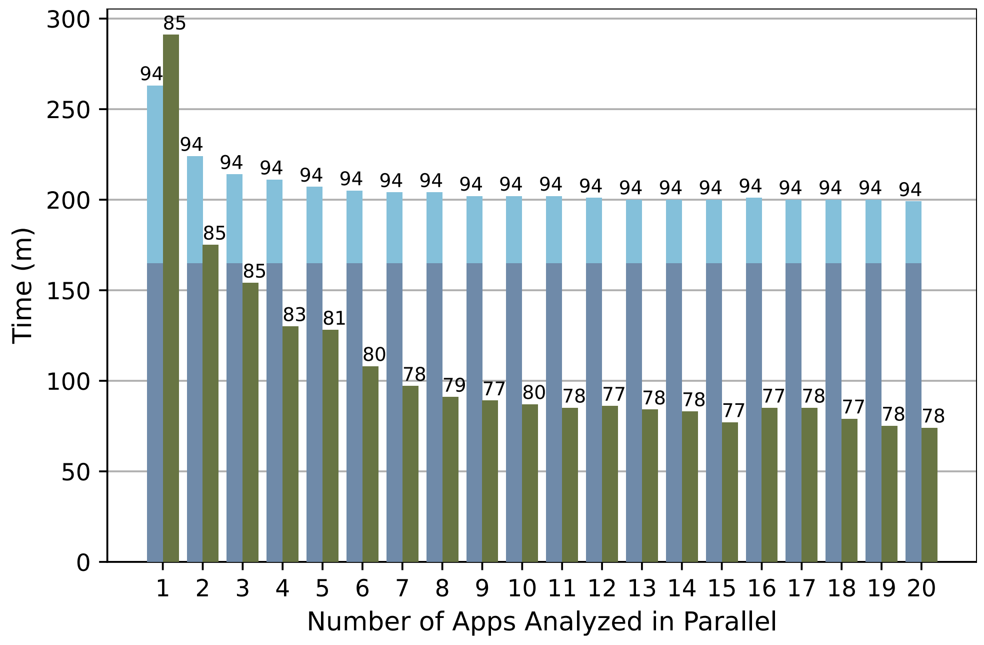


FIGURE 4.11: Analysis time of T-Recs and FlowDroid with different number of apps analyzed in parallel. The blue bars represent T-Recs' results, and the green bars represent FlowDroid's results. The labels on the bars show the number of successfully-analyzed apps.

one to 20. FlowDroid was also tested for comparison. The analysis time may vary depending on the order of the apps to be analyzed. In this study, the tools analyzed the apps in alphabetical order by the app's name.

Figure 4.11 shows the result. The blue bars show the analysis time taken by all phases of T-Recs. The dark blue parts represent the time for the parser, the instrumentator, and the app exercise. The light blue parts show the reconstructor's time. For example, 98 minutes for #apps = 1, 59 minutes for #apps = 2, and 34 minutes for #apps = 20, which is the shortest. The bars' labels represent the number of successfully-analyzed apps. The maximum is 94 because T-Recs fails two apps in phases prior to the reconstructor, and the reconstructor processes only 94 apps. On the other hand, the green bars indicate FlowDroid's result. For example, 291 minutes for #apps = 1, 175 minutes for #apps = 2, and 74 minutes for #apps = 20, which is the fastest. However, as the bars' labels show, the number of successfully-analyzed apps decreases as the number of apps analyzed in parallel increases. FlowDroid's maximum number of successfully-analyzed apps is 85 because FlowDroid fails 11 apps regardless of the RAM size. Hence, the maximum number of apps that FlowDroid can analyze in parallel without causing failure is three, and the time is 154 minutes.

For the results in Table 4.9, the number of apps analyzed in parallel was determined based on the maximum number that would not cause the analysis to fail due to lack of memory. T-Recs' reconstructor was executed in 20 threads, and FlowDroid was executed by analyzing three apps in parallel.

T-Recs' and TaintDroid's App-Runtime Overheads

In dynamic analysis (i.e., T-Recs' app exercising phase and TaintDroid), the app-runtime overhead affects the operation delay, which in turn makes the analysis time longer. If the app exercise time is too short, the app may be terminated before a leak occurs, and the leak

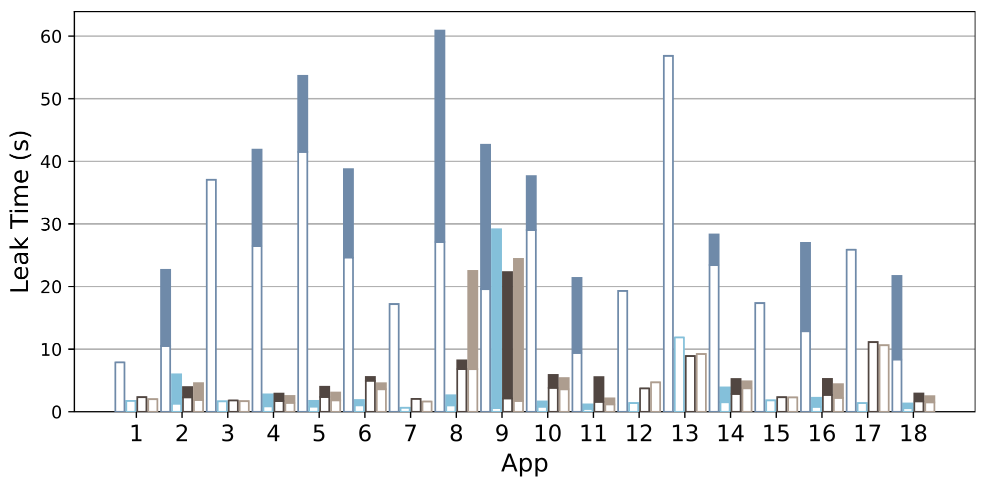


FIGURE 4.12: Time for apps to launch and cause leaks on Pixel 3 with and without T-Recs and Nexus 4 with and without TaintDroid, which are represented by the dark blue bar, the light blue bar, the dark brown bar, and the light brown bar from left to right, respectively.

would not be detected. Therefore, the app-runtime overheads of T-Recs and TaintDroid are investigated by measuring the time for apps to launch and cause the same leaks with and without the tools. Pixel 3 was used with and without T-Recs, and Nexus 4 was used with and without TaintDroid to compare with T-Recs.

Figure 4.12 shows the leak time of the 18 apps that cause one or more leaks. Pixel 3 with and without T-Recs and Nexus 4 with and without TaintDroid are represented by the four bars from left to right for each app. A transparent part of the bar indicates the time for the first leak in the app, and the colored part indicates the time for the last leak in the app (i.e., the time for occurring all the expected leaks in the app shown in Table 4.6). In a total of the 18 apps, T-Recs took 578.1 seconds; Pixel 3 without T-Recs, 74.9 seconds; TaintDroid, 104.0 seconds; and Nexus4 without TaintDroid, 113.1 seconds. T-Recs is 7.7 times slower than the original Pixel 3 and 5.6 times slower than TaintDroid. Using TaintDroid is faster than not using TaintDroid, indicating that TaintDroid has no overhead, and the result is consistent with the original paper [54], reporting that TaintDroid has negligible perceived latency with real-world apps. T-Recs took the longest time for app number 8, which was 60.9 seconds. TaintDroid took the longest time for app number 9, which was 22.3 seconds. The T-Recs' longest time is 2.7 times larger than TaintDroid.

The app-runtime overhead of T-Recs is caused due to the instrumentation, which is a trade-off for the tool's device independency. Preparing Android devices is easier for T-Recs than TaintDroid. Hence, T-Recs users can run the app exercising parallelly on multiple devices to shorten the analysis time. For example, assuming that the analyst sets the same exercising time for every app (i.e., 61 seconds per app for T-Recs and 23 seconds per app for TaintDroid), T-Recs' time would be shorter than TaintDroid if three or more devices were used in parallel for T-Recs. Hence, the app-runtime overhead of T-Recs is considered to be acceptable.

The time for apps to be installed and uninstalled, which affects the analysis time, is measured. Table 4.10 shows the total time for 90 apps with T-Recs, Pixel 3 without T-Recs, TaintDroid, and Nexus 4 without TaintDroid. The experiment excludes two apps that T-Recs failed to complete the analysis and four apps that TaintDroid failed. The uninstallation times are the same for Pixel with and without T-Recs. It barely changes for Nexus 4 with and

TABLE 4.10: Time (seconds) for apps to be installed and uninstalled on the devices with and without the tools.

	Pixel 3 w/ T-Recs	Pixel 3 w/o T-Recs	Nexus 4 w/ TaintDroid	Nexus 4 w/o TaintDroid
Install	497	396	1714	1571
Uninstall	36	36	119	114

TABLE 4.11: #apps successfully analyzed, #apps failed, and the analysis success rate for each tool in the privacy leak detection.

Tool	#apps succeeded	#apps failed	Success Rate
T-Recs	94	2	98%
TaintDroid	92	4	96%
IntelliDroid	72	24	75%
FlowDroid	85	11	89%
FlowDroid$_{IC3}$	29	67	30%
Amandroid	12	84	13%
DroidSafe	0	96	0%
DroidRA$_F$	4	92	4%
DroidRA$_A$	0	96	0%
DroidRA$_D$	0	96	0%
IccTA	8	88	8%

without TaintDroid. On the other hand, for the installation times, although using T-Recs is 26% slower than not using T-Recs, the T-Recs' result is 3.4 times faster than TaintDroid. This is mainly because T-Recs' device, Pixel 3, has a higher processing performance than TaintDroid's Nexus 4. The result demonstrates the device-independency advantage of T-Recs, in which the analyst can use new smartphones with high-performing processors.

Analysis Success Rate

Table 4.11 shows the results. T-Recs' success rate is 98%, the highest among the compared tools. It failed one app in the instrumentation phase and one app in the installation phase. No type-conflict error occurred during app execution, and the success rate of the logger is 100%. The reconstructor also succeeded in detecting information flows.

FlowDroid's success rate is 89%. It stopped during the analysis due to some runtime errors with 11 apps. All failures occurred in the call graph construction phase before the flow detection process. These failures occur regardless of the taint source and sink definitions. FlowDroid$_{IC3}$ failed with 67 apps, mostly due to IC3 failures (e.g., exceeding the timeout). TaintDroid failed to install four apps due to incompatible SDK versions. TaintDroid uses Android 4.3 at the latest, and all apps that do not support this version cannot be analyzed. IntelliDroid's success rate is 75%, and the other tools' success rates are diminutive.

4.4.4 Success Rate of Essential Phases in Varied Dataset

This section focuses on the tools' essential phases that are independent of tracked data to obtain the upper bounds of the tools' analysis success rate in general. Section 4.4.4 explains the essential phases, which vary from tool to tool, and Section 4.4.4 presents the result.

Compared Tools and Setup

Since IntelliDroid, FlowDroid$_{IC3}$, Amandroid, DroidSafe, DroidRA$_F$, DroidRA$_A$, DroidRA$_D$, and IccTA detect almost no leaks (Table 4.6), and their success rates are less than 80% (Table 4.11), this section excludes them. Hence, this section compares T-Recs, FlowDroid, and TaintDroid. They may fail for reasons other than those described in Section 4.4.3. However, hidden errors could not be detected without knowing the tools' details. Also, errors in information flow tracking could depend on tracked data. In dynamic analysis, app exercise also depends on data being tracked. Therefore, this section focuses on the tools' essential phases, independent of taint source and sink definitions.

Since T-Recs' instrumentation and app-installation failed (Section 4.4.3), the percentage of successfully instrumented and installed apps was examined. The device used was Pixel 6 (Android 12), which was the latest device available at the paper was written and is much newer than TaintDroid's Nexus 4, highlighting the device-independency advantage of T-Recs. Considering that FlowDroid failed regardless of the taint source and sink definitions (Section 4.4.3), FlowDroid was executed without taint source and sink definitions to examine the percentage of analysis failures that occurred in the call graph construction phase. No timeout was used. Since TaintDroid's app-installation failed (Section 4.4.3), the percentage of apps that are successfully installed to TaintDroid was examined. TaintDroid failed because of the apps' supported SDK versions, which can be acquired by investigating the app files, but other factors may cause the installation failures. Therefore, the apps were actually installed one by one to TaintDroid without launching them.

Results

The rate of successfully-processed apps for each Android codename is shown in Figure 4.13. The Android codename (e.g., 1.0, 1.1, C, and D) indicates the minimum SDK version supported by an app configured by the app developers. The result shows that T-Recs evenly supports the 16 codenames (i.e., 28 SDK versions). T-Recs achieves at least 86.3% for any version, and the average is 96.6%. FlowDroid also achieves at least 87.5% for any version, and the average is 95.6%. On the other hand, TaintDroid fails for apps newer than version J (i.e., Android 4.3), and the average is 62.5%. TaintDroid is not applicable for apps developed for Android 4.4 or later versions after 2013. In this evaluation, T-Recs took 276 hours; TaintDroid, 116 hours; and FlowDroid, 108 hours.

4.4.5 ID Leak Detection in Popular Apps 2021

Recently-published popular apps were used to consider a more up-to-date situation than Section 4.4.3 because access to the hardware identifiers has been restricted since Android 10.

Compared Tools and Setup

This section compares T-Recs, FlowDroid, FlowDroid$_{IC3}$, Amandroid, DroidSafe, DroidRA$_F$, DroidRA$_A$, DroidRA$_D$, and IccTA. It omits TaintDroid and IntelliDroid because TaintDroid's success rate is low for newer apps (Section 4.4.4). The app set used in this section contains 139 apps (88%) with minimum SDK versions of 19 (i.e., Android 4.4) and greater, which cannot be analyzed by TaintDroid and IntelliDroid. As a result, T-Recs was only compared with static analysis, which was disadvantageous for T-Recs because of the coverage difference,

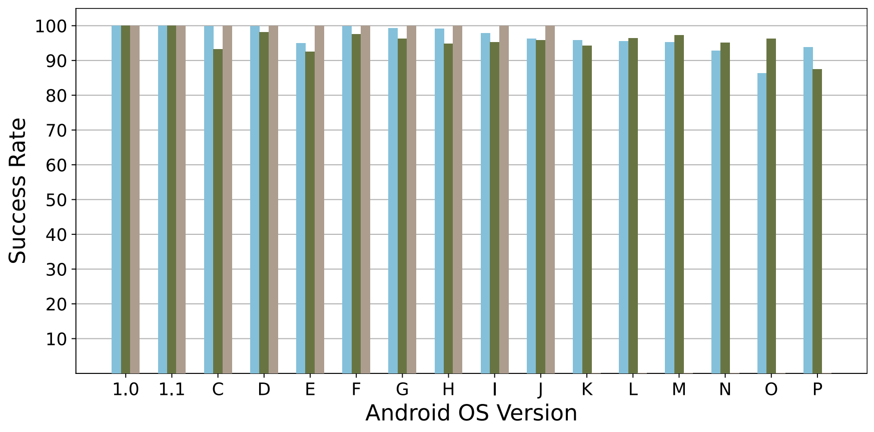


FIGURE 4.13: Success rates for the varied dataset. T-Recs is left blue bars, FlowDroid is center green bars, and TaintDroid is right brown bars.

TABLE 4.12: #apps and #leaks in parentheses detected in the apps from 2021. "Overlap" indicates #apps and #leaks detected by both T-Recs and the other tool.

	T-Recs	FlowDroid	FlowDroid$_{IC3}$	Amandroid	DroidSafe	DroidRA$_F$	DroidRA$_A$	DroidRA$_D$	IccTA
	55 (400)	60 (214)	4 (6)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
Overlap	-	3 (8)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)

and static analysis should detect more leaks. However, showing that T-Recs can detect leaks in newer apps is still valuable.

T-Recs launched and waited for each app on Pixel 3 for two minutes with no exercise operation based on the assumption that ID leaks swiftly occur as well as the evaluation in Section 4.4.3. A one-hour timeout is used per app for each tool.

Target taint sources are identifiers, including Build.SERIAL, MAC address, Android ID, Google Advertising ID, Instance ID, and Globally-Unique ID [128] in addition to the taint sources used in the privacy leak evaluation (Section 4.4.3). Taint sinks are the same as the privacy leak evaluation. As described in Section 4.4.3, this evaluation also uses the default definitions of taint sources and sinks for DroidSafe.

Results

Table 4.12 shows the number of apps and leaks detected by the tools. T-Recs and FlowDroid detect leaks in 55 and 60 apps, respectively. FlowDroid$_{IC3}$ detects six leaks in four apps, which are also uncovered by FlowDroid. The other tools generate no alerts.

T-Recs detects leaks in fewer apps than FlowDroid, as expected. In contrast, T-Recs detects 400 leaks, which is larger than FlowDroid's result. Based on the result in Table 4.6, T-Recs generates numerous FP leak alerts as well as TPs for leak-causing apps. Therefore, some of the 400 leaks could be FP. At the same time, however, Table 4.6 shows that T-Recs produces no FP for no-leak-causing apps. Hence, all 55 apps detected by T-Recs in Table 4.12 are likely TP. Interestingly, the detected apps and leaks overlap slightly between the tools. The result shows that T-Recs can track information flows and detect ID leaks, primarily undetected by FlowDroid, in recently-developed apps from the Google Play Store.

On the other hand, Table 4.12 shows that FlowDroid detects 206 leaks that T-Recs does not detect (i.e., among 214 leaks detected by FlowDroid, the overlap is only eight leaks). It was confirmed that 205 leaks are caused by codes not covered by T-Recs. In contrast,

TABLE 4.13: Analysis time for the ID leak detection.

Tool	Time
T-Recs	14 hours 35 minutes
FlowDroid	19 hours 58 minutes
FlowDroid$_{IC3}$	22 hours 18 minutes
Amandroid	122 hours 33 minutes
DroidSafe	4 hours 33 minutes
DroidRA$_F$	149 hours 44 minutes
DroidRA$_A$	149 hours 45 minutes
DroidRA$_D$	149 hours 45 minutes
IccTA	42 hours 1 minute

TABLE 4.14: #apps and #codes in parentheses in which the five code types are found and whether FlowDroid detects the leaks caused by the five code types. The row "any" gives #apps and #codes in which at least one code type is found.

	T-Recs		FlowDroid		
Type	flow	leak	positive	negative	incompleted
1	1 (2)	1 (1)	0 (0)	1 (1)	0 (0)
2	2 (2)	0 (0)	0 (0)	0 (0)	0 (0)
3	40 (95)	25 (89)	0 (0)	20 (71)	5 (18)
4	43 (177)	16 (59)	0 (0)	12 (40)	4 (19)
5	22 (24)	6 (21)	0 (0)	4 (6)	2 (15)
any	52 (223)	29 (110)	0 (0)	24 (83)	5 (27)

the other leak is caused by codes within T-Recs' code coverage. However, T-Recs does not detect the leak because of the leak's sink. This sink is a writer that outputs data not to the network but to a file. Hence, detecting the leak is FP. T-Recs identifies that the information flow's destination is a file and does not detect it as a leak. Overall, FlowDroid detects leaks undetected by T-Recs principally because of the difference in their code coverages. The maximum, minimum, average, and median of the T-Recs' code coverages were 28.9%, 0.05%, 6.1%, and 4.6%, respectively. Note that 16 apps with zero code coverage were excluded from this calculation. In contrast, FlowDroid can analyze the whole code of each app, which is a trade-off for accuracy, and Section 4.5 further discusses this.

Table 4.13 shows the analysis time taken by each tool. T-Recs took 14 hours and 35 minutes. The parser and the instrumentator took five hours and 13 minutes; the app exercise, seven hours and 42 minutes; and the reconstructor, one hour and 40 minutes.

Tracking Ability for ICC- and Reflection-Related Flows

Similar to Table 4.8 in the privacy leak evaluation (Section 4.4.3), Table 4.14 shows T-Recs' and FlowDroid's results for the five types of code related to ICC and reflection. It shows the number of apps and code points where the five code types are found (second column *flow*) and where the related leaks are detected (third column *leak*). It also shows whether FlowDroid detects the leaks (fourth, fifth, and sixth). The result shows that all five code types are found, and also leaks related to four code types are found. In contrast, FlowDroid detects none of the leaks. FlowDroid fails to complete the analysis for some of the apps, as indicated by *incompleted*. The results show one of the reasons why T-Recs' and FlowDroid's

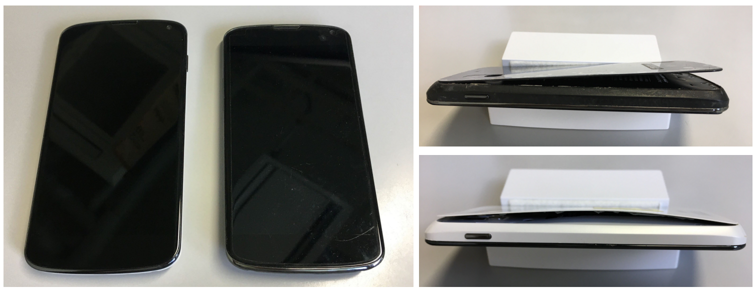


FIGURE 4.14: Two Nexus 4 devices used to run TaintDroid and IntelliDroid. Their batteries have been swollen in approximately one year of use.

results barely overlap. The results emphasize the importance of tracking ICC- and reflection-related flows as the flows appear in the recently-published real-world apps.

As well as Section 4.4.3, this experiment was conducted after the results in Table 4.12 were obtained, and re-exercising the app was unnecessary. It is also discussed in Section 4.5.

4.4.6 Ethical Considerations

In the evaluation, the experiments were carried out only by my team. Android devices used in the experiment were bought and prepared for the experiments, and no actual personal information were used. Respect for app publishers is also considered. The bytecode instrumentation targets only code that does not change any data sent to remote servers so as not to affect the publishers' properties. Additionally, the results were carefully used only to evaluate the tools and were not used for other purposes.

4.5 Discussion

As mentioned in Section 4.4.3 and Section 4.4.5, T-Recs can re-execute the taint analysis (i.e., the reconstructor) without the app exercise after a new feature is added to the taint analyzer. In the privacy leak detection (Section 4.4.3), the reconstructor took 34 minutes, which is 17% of the whole. The reconstructor took one hour and 40 minutes, which is 11% of the whole in the ID leak detection (Section 4.4.5). Note that these times were measured after the new feature (i.e., the ICC- and reflection-related-flow counter) had been added. A large amount of time is saved because T-Recs' analysis time depends mainly on app exercise time. As another example, in the DroidBench evaluation (Section 4.4.2), the exerciser took four hours and 43 minutes (i.e., 95% of the whole). In addition, the app exercise time can be extended when the analyst targets other than privacy leaks because this paper currently focuses only on privacy leaks, which tend to take a short time to occur. Therefore, being able to re-execute the taint analysis without running the app exercise is a significant advantage of T-Recs.

The evaluation shows that T-Recs has higher usability than TaintDroid in terms of ease of setup. TaintDroid is only available on Nexus 4 or older devices. Accessibility of the devices is extremely poor, and also, devices have a limited lifespan due to, for example, battery swelling (Figure 4.14). T-Recs, on the other hand, allows analysts to freely select devices such as Pixel3 and Pixel6, depending on each evaluation's environmental requirements (e.g., Android OS version). These devices are readily available, and the analysis can be started speedily by simply connecting them to a computer.

In the evaluation, privacy leaks are the main target and can be easily triggered. The core of this paper is to perform a runtime-data-utilized taint analysis outside the Android device, and improving code coverage of real-world apps is out-of-scope. As long as there is a code coverage problem with dynamic analysis, it is unrealistic to replace static analysis with dynamic analysis in some cases, and the analyses should be selected according to the user's purpose. Static analysis should be used in exchange for higher verification costs when code coverage is a priority. T-Recs should be used when low verification cost (i.e., accuracy) is a priority.

In the DroidBench evaluation, the categories covered by the compared tools [78] were selected, and others, such as native code and inter-app communication, are out-of-scope. It is believed that T-Recs can be used in combination with existing tools, such as JN-SAF [50] for summarizing flows in native code because T-Recs performs the taint analysis on the server.

The manual analysis of the network dumps explained in Section 4.4.3 is limited, which could affect the accuracy of the data in Table 4.6. Only plain texts of the target information, their names, and transformed data found by T-Recs and TaintDroid were searched in the network dumps. Therefore, some leaks could be missed, for example, leaks caused by an app performing complex encryption on IMEI without being detected by the tools. Such mistakes affect the following. First, there would be more FNs in T-Recs' and TaintDroid's results. However, the impact is considered to be small because modifying both tools' results by the same amount does not change the conclusion that T-Recs generates fewer FNs than TaintDroid. Second, if a missed leak had been counted as the unsure in FlowDroid's result, the number of unsure leaks would have decreased, and the number of TPs would have increased. However, detecting such encrypted leaks requires reverse engineering of the app. The cost is high, which is consistent with the argument that the verification cost of the unsure leaks is high in Section 4.4.3. Therefore, the effect of missing leaks is considered to be acceptable.

Apps may detect code rewriting and stop running to protect the app developers and users [129]. When analyzing such apps, the logger must be integrated into the Android OS instead of the app bytecode instrumentation. As a result, the logger would depend on the Android OS version, the same as the existing dynamic analysis systems. However, compared to implementing the taint logic itself, implementing only the logger would be less expensive and more practical.

Whereas TaintDroid performs variable-, method-, file-, and message-level tracking, T-Recs does not track inter-app messages and does not keep tracking tainted content in a file across different runs. Although this was not a problem in the evaluation, it could depend on the information being tracked. As well as TaintDroid, T-Recs disregards implicit flows [89].

4.6 Related Work

Various tools of static taint analysis for Android apps have been developed and assessed in the community [77, 78, 130]. Mordahl et al. [130] examined configurations in FlowDroid and DroidSafe. Pauck et al. [77] evaluated static taint trackers: Amandroid, DIALDroid [19], DidFail [35], DroidSafe, FlowDroid, and IccTA. They excluded unavailable and unsatisfied tools, such as SCanDroid [131] and DroidInfer [49], for competitive comparison. Zhang et al. [78] compared FlowDroid combined with IccTA, Amandroid, and DroidSafe under the same setup. They also included DroidRA, an instrumentation-based approach targeting reflective calls and used in combination with FlowDroid, Amandroid, and DroidSafe. This paper followed the study and used the same tools and configuration parameters because the exact set of used benchmark applications and the answers are available [115].

RAICC [34] is one of the latest tools targeting ICC. Specifically, RAICC targets "atypical ICC methods", which allow to perform an ICC while it is not its primary purpose [34]. Since the 158 apps in DroidBench do not contain "atypical ICC methods", RAICC instruments none of the apps in the evaluation (Section 4.4.2). Its developers confirmed the result and mentioned that it is intended for RAICC to instrument no DroidBench apps. On the other hand, this paper mainly focuses on addressing the inaccuracy of current taint analyzers against DroidBench, which contains more common cases than "atypical ICC methods". Also, Barros et al. [39] developed a static analysis technique for handling ICC and reflective calls precisely. Their approach is implemented for Java and requires the target apps' source code. However, when analyzing apps from the Google Play Store or third-party markets, their source code is not usually available.

As native code is being more frequently used in apps, researchers have been developing new static analysis techniques targeting native code, such as JN-SAF [50] and JuCify [52]. JuCify unifies call graphs of native code and bytecode, and the result can be used by FlowDroid and other static analyzers that do not support native code. Also, CTAN [51] improves JN-SAF. Furthermore, μDep [53] empowers static taint analysis tools, such as DroidSafe, by performing static and dynamic analyses of native code to model the tainting behaviors of native code. As Section 4.5 explains, native code is out-of-scope of this paper.

There are more tools [62, 63, 65, 64], which perform the bytecode-level dynamic taint analysis for Android apps other than TaintDroid. Although, in comparison with static taint analyzers, dynamic taint trackers have been barely reviewed in the community except for TaintDroid. The taint tracking module of ARTist was given by the authors. However, the authors mentioned that the module is aged and requires quite some adoptions to be used with ARTist. Therefore, the tool was not used in the evaluation. TaintMan was obtained from the authors. However, difficulties were encountered in deploying the tool, and the tool was omitted. TaintART is not publicly available, and the authors did not reply to a request. Since native code is out-of-scope, OS-level trackers [67, 69] were excluded. For these reasons, only TaintDroid was chosen for the evaluation.

Recently, researchers developed hybrid analysis techniques such as targeted execution [59] and program slicing [60], which can assist taint trackers in detecting more leaks. IntelliDroid performs targeted execution, which is efficient when attempting to run a specific code path. However, it depends on static analysis and inherits the drawbacks of inaccurate models. It was evaluated in the DroidBench evaluation in Section 4.4.2, and its result only slightly increased from TaintDroid. Besides IntelliDroid, Harvester [60] can improve TaintDroid by triggering malicious code. However, Harvester was omitted because the user must coordinate its target logging points (i.e., not wholly automatic), and also, Harvester is not publicly available.

4.7 Summary

This chapter presented a usable taint tracker called T-Recs, which detects information flows by recording and reconstructing the app execution. T-Recs addresses the limitations of current dynamic trackers, which are the dependency on the analysis environments and the reanalysis cost. T-Recs was implemented and evaluated with 158 apps from DroidBench, 96 and 158 popular apps from the Google Play Store, and SDK-version-varied apps randomly collected from the Google Play Store and Anzhi. The results show that T-Recs is making steady progress. While T-Recs gives overwhelming results, static taint trackers may keep establishing a strong presence in mobile security and privacy studies because they are easier to use, more scalable, and wider in coverage. T-Recs, on the other hand, should be used to leverage the advantages of runtime data. For example, if a study focuses on apps' behavior changes depending on the apps' and their servers' configurations, utilizing runtime data is more realistic than analyzing only the apps' code statically. T-Recs has been made publicly available at `https://github.com/SaitoLab-Nitech/T-Recs`.

Chapter 5

Conclusion

This thesis proposed two approaches named VTDroid and T-Recs based on the idea of utilizing the app's runtime data to improve the taint analysis.

Chapter 1 Android OS occupies 70% of the total mobile OS market share in 2023. The official market, Google Play Store, currently provides 2.6 million apps downloaded and used in users' day-to-day activities on their devices, always connected to the internet.

On the other hand, the need to protect user privacy has been increasing. Data protection regulations, such as COPPA, CCPA, and GDPR, have been implemented in the past two decades. Google has also made changes to data protection mechanisms on the OS and policies in the Google Play Store.

It is also essential to uncover how well app developers and third-party SDK providers follow the rules to protect user privacy. Researchers have investigated real-world apps and found many non-compliant, policy-violating, and protection-mechanism-circumventing behaviors. Taint analysis techniques have been actively developed and utilized to detect such suspicious behaviors.

Chapter 2 Chapter 2 described the fundamentals for understanding taint analysis for Android apps. A taint analysis system must take information flow types into consideration to avoid overlooking information flow. In addition, Android apps have various unique features, making the taint analysis more difficult. In order to balance precision and recall in these situations, current mainstream analyzers are bytecode-level trackers and target only direct data flows.

Chapter 3 Chapter 3 explained VTDroid, designed to make it difficult for apps to evade taint tracking by neutralizing uncomplicated ATA techniques not specific to a particular ATA technique. A series of ATA techniques has been demonstrated on the Android platform. They are only a few lines of code each and could be introduced into apps with obfuscator tools by app developers to protect their apps against a taint analysis. However, there are only a few counter approaches against ATA techniques, which are only partially effective against the variety of ATA techniques.

First, this chapter characterized the ATA techniques by the four types of information flow. Then, it proposed value logging and matching that propagate taint among registers based on their data values, in addition to the traditional bytecode-level tracking. VTDroid was evaluated with newly created test suites and real-world apps compared with TaintDroid, CTT, and FlowDroid. The results demonstrate that VTDroid tracks more information flows resulting from the ATA techniques and generates fewer FPs than CTT. Therefore, VTDroid makes handling ATA techniques more realistic than CTT in real-world app analyses. VTDroid's performance demonstrated in this chapter should be an indicator for researchers to determine whether they are concerned with ATA techniques in their studies.

Chapter 4 Chapter 4 described T-Recs, a runtime-data-utilized taint tracker that solves the current situation of no tracker that can analyze apps reliably. The community needs a reliable taint tracker for analyzing real-world apps. Researchers recently tested popular static taint analyzers and concluded that the tools are inaccurate and cannot be used for analyzing real-world apps dependably. On the other hand, researchers examined a famous dynamic taint tracker, TaintDroid, and pointed out that TaintDroid is the most difficult to set up compared to the static analysis tools they audited. Also, TaintDroid depends on specific devices and versions of Android OS released in 2013, narrowing down the scope of analyzable apps. Other dynamic analyzers are not effortlessly usable.

T-Recs addresses the issue. It records and reconstructs the app execution and performs taint analysis on an ordinary computer (e.g., a computer running Linux), not depending on Android OS. T-Recs' accuracy, analysis time, and success rate were evaluated in privacy leak detection compared to currently available taint analyzers, which are FlowDroid (w/ and w/o IC3), Amandroid, DroidSafe, DroidRA, IccTA, and TaintDroid (w/ and w/o IntelliDroid). The evaluation involved 158 test cases in DroidBench, 254 popular apps from the Google Play Store in 2016 and 2021, and 39,480 SDK-version-varied apps from the Google Play Store and Anzhi. The results show that T-Recs outperforms the compared tools in detection accuracy. T-Recs also achieves reasonable analysis time, app-runtime overhead, and success rate. T-Recs is making steady progress.

While T-Recs gives overwhelming results, static taint trackers may keep establishing a strong presence in mobile security and privacy studies because they are easier to use, more scalable, and wider in coverage. T-Recs, on the other hand, should be used to leverage the advantages of runtime data. For example, if a study focuses on apps' behavior changes depending on the apps' and their servers' configurations, utilizing runtime data is more realistic than analyzing only the apps' code statically.

Data and code availability VTDroid and T-Recs have been made available to the community. VTDroid is released at `https://github.com/SaitoLab-Nitech/VTDroid`. The ATA test suite for privacy leak detection is available at `https://github.com/SaitoLab-Nitech/ATATechniques`. The ATA test suite for suspicious validation detection is also available at `https://github.com/SaitoLab-Nitech/ATATechniques_SuspiciousValidationDetection`. T-Recs has been made available at `https://github.com/SaitoLab-Nitech/T-Recs`. The other datasets used in this study are available from the author upon reasonable request.

The regulations, market policies, and protection mechanisms will be reformed in the future, and researchers should keep examining apps and libraries. VTDroid and T-Recs should be promising tools that empower researchers to analyze apps in the future.

Appendix A

Hashes of the Analyzed Apps

A.1 Apps Used in the VTDroid Evaluation

This section gives hashes of the three sets of apps used in evaluating VTDroid explained in Section 3.5.

A.1.1 Popular Apps Collected from Google Play Store in 2016

Table A.1 shows hashes of 30 apps used in Section 3.5.4.

TABLE A.1: Hashes of 30 apps collected from Google Play Store in 2016.

385bd2a3e0e6c99e175fe131bcad9db89e60c341538556aa784c7a8fe5990422, 41126e3999cb4d7a7910716260a8255d7754e7ddc945c68de6a089b1c0dd8ccc,
c94539ba3e4777a1346bf595d74365fd6d5de6d6457d9b079f8b2d3fda3cab9a, 328663f3ee60c1ab9e0c915892634b9bc80c2bb1142f47d74d39a0c4eba118f1,
1d6bfdb5bfe601b5a35cdd0c7be2320d5cd46180584fd51243b292626990dec0, 9c4e0807ac1d830786dcb9b6731fb08eb047d81f1a42711471182205b8ea72df,
f33e3fd3e3a020addb9c7edfdf0b3b13bc382221dc1c967a81a7de702dc239ad, 2183e6282243269d7ab25a31b5f0b1fcfd15d73b0e24890fb5bace8c19657f53,
af7a3cc40ffb6930398048a86f1a2c4cd1368c966e5826534a7118a89d45eb28, 2ab6ea68266066f476bc251ebfc5ac7e510bde98d3706d74d3feb5bc9dfce0eb,
4e7abb31e72a286447ccec93ea5604bbbbc5e43f273d3301f312c564d4071c02, 235010d4454a3dce9b334f99a26ddc7d4123737363a43823c19d96fb970ba00f,
7600e01d616401987dbeb194adabe4bed0906fcd0f2063eb34f5a7f3cc6441dc, efb39a8b64400c285ff41c90e0abb734c2f76059c8ae9b6631e533c841c7d7f6,
eae11c32fadd6fb3eb171d1cf78aec84ec537a35bc15648b1a7b91b4af09deb2, a14c4158898656abc08bcbf43df9299c5dec984068460d5826f2e8ef5f0b7c75,
a16ed39c64a7cfb692a1b42757ce5bb85dcf60355ca4b98109e3445fdf5d23bd, 668047b8393391a3682e5ef429164124f2990f39f9dc3c9b130340862fd87436,
44ac462f846cffcbe3d3156d00c9837a30228b12fa4b2287bfdf1767edbda3e7, 401722b99fe626a1ba6bbed8f5809135dbed706c061315d308e7284e4dbb91a7,
9d919ba99629b8a061aab253135c186167b2ea6123d7d3830654869cafccd5f4, 8cf04193f124693a5643dcbaf1f0b969f9a481c6ff749acad8a1bd8e15fdb1ac,
554730324c46d55c0ed84c74449b1faa9694f59ecf9c9dcaa161a0e6cbcbd7ae, fcef2e5635cc0baebbcb9d4cad61b2afec163a9f8da7dc23e728e2e2912990f9,
98f7cbd948ddf41b112579600535da71f748394c1a6f61e9ebfe6a158f5291ed, 7905242ba51fe30824b2cc3ccced60c63a1bf49a2065f7024a399c0e51572e68,
a44626cbdc5d85d230d3afd7be29dd660d53e911e88fcd44e87e37145d462183, d1eb0e7e4faf16832adcbf3f2ac3341e3a4e8c98b22ee24b9b52526f0c11b82a,
548ed9f19181381c0440a1ef07937211a253799a32d92f77300967525da63a7e, 703b0f193ebc9713da8d56dcc3bf668683690e92d0ed7ffa8ce9a85f9ba6d405

A.1.2 Popular Apps Collected from Google Play Store in 2021

Table A.2 shows hashes of 277 apps used in Section 3.5.5.

TABLE A.2: Hashes of 277 apps and games collected from Google Play Store in 2021.

93743bdd9ec9919446159fbdc4716505fb13776a5423572f85cd31887fea6057, aea9687feef84d6c5c45ba6828520b45d9320a50b8acbb3e00898b29b91ac751,
99ed3082155257df8def811c47b6fa1980a31750592dce33b7a22dc0fad50e45, 4db06f56dcfcec21cc1808e8954bf5272bbef4d2237a822963324d0d76b62a5c,
e53b89177e697eff71776ab360727afee73f58e4492cd758a8cb47d3618bdc12, 3ffe2a52eda8f4eb3cc90d599aa7252bfe0759d1e7e65f23221ef007797fa947,
882ca0bd7bc58d380ca3fcae3510891eeea4475e37728a065e43d5d9013a5fed, 7680b4db799fccbfd639efefb6856b4e83a20c58f0733a52abcd565bec76ee85,
4c5e1236595c81a04dbee97d374109e4f4df9cec990bc514bb2098e364f73e00, e39ad0baee1d61246e17f7d3d213cf2c65cf3ee41c6851230f13d50224bcf0e4,
bba3d239fddd7bb1bc7b18204cfc0380a1ce519ffd08716cd65184e8cf86bd66, 8c16ffae75fa7609626feb15b317486286bc752fb2dd9e05e3209758054133c4,
976f247bf4dd445d7b51f5db2daf439e1eb08297b6a4432332425f7cbd6ec27e, a595aff30bb60a8a8b583af6998fd4d13edc9ad4f6bff19b194d1c5d6d459e56,
dd7a2833705d79867531ec660e1c9f441156fc61381f292f13bcad6b9677f6e2, 89faf29f3d81ded7e81d0adde7cac501f0ad645d68a420866f35f31a7a50f292,
f0e1f423dfb50a5a0c7135fded2fdc777e1e2fe14c9eb5df099271e9b2e534bd, 0346f2e43613fe50be16c9fce9ad3ffa689a50773c353f19171a3e51704c4809,
d318a6994b74960bb85e3f454af59b2f03605db83f19732c6d13928527848c58, d5f4ad05fbfafe15bfd5b5d6c2a294e0a9c250211a026fa090977f535d30eac3,
2f5ff3459c73a2d168e8cf5f3339047b346dc82a044e262dd6820d8cbafc87b6, ce5365bc90d3b750a85ba5fb8eadff09c22a9b70d200ef77aa8c0970f6c31d16,
b66e0b731dcddd0f167bdfde4c9b1eae31a86f242cb9f0be37988a7478eee521, e67f019b3aba4e8388837da735866b8e241882a2128a2db6d78a707d54ab31d0,
fe762b1bd71b2a9ee80cd1a85e189c21078b069fa921dd455e0286cf875abc10, 1b20f8305956e0ee245b488b2cb451746efb44793a00e03bb346280196c4c974,
ea9adc48032442f3779c04c084b2860d44cb16b5a33397b6c78ff59cc75b9fe8, f9bfa1680d1b62178ab27bf1d6829cafd9e1af0097d7ee22b3677bf666e7bde6,
f998a13609e1a2ca0ab2eb76180dc3ed8970dee2f3f72bccf7a3e1fa9742c226, db2c734d9041878ac99f32023b755a0af6566348e7925c13267588edc734d967,
a3a434d95f0fec2e543328c85cd3fef412dfc465169b41979762ddf507f90680, 9c463392057c86f0ccce39e93ffeb8fb2f225ecec31cb84c2da8b60884245efc,
c808ef7e416c7db772840e66a8416e07e13a5b08d2e064c556f65079fac9e4a5, f28ea1b9f085339b4c664562377e7926d6a999ca81b768fa8286c53b5be90fde,
5ceab8764de790b3550c653d736d483b56afc459a7a5e49bffacd786ac7e00ca, e352f9d9d74bdd0c3578296ae4e80dd7b3722d24568a44ea5889e9d5e13e4477,
20c224b1b12713b743326d89c4294bb54c6e47605a14395bf548a7672e42a3c3, 66044239eb7317c34bf1ecd8855bc26497dfdb32de68cf14b62f6916d74d1835,

99dc4b979f8499412a0e0f06d963f077f852b08bf066fbe28636105595da9d4f, 46a0aacacd6c403bafc692e55e71cf58ebf6a9b156dc5cf159b5a29c97f76d29,
b39f84e7146afed6074f96b90f56a2ef621b26e835c9b2c1dc72f7b43be594a5, 91eec64749ac26af48a94cc352534dc658b5394d395ff26f68e1091bb0caa25d,
25678f286172670a72478b3418c6d99689923e47ecc5f5aa888782546e1a0233, 0b9f81c77fb6161e0c69ca08fae8e63d59bf07216d4f5cb559909a8717ef6ed1,
c5e41c76f0074ce833219d97bbd16fe2caaeedd1fed92a16bf4e53c42ea7ceee, ee2dcd56258aee6f7e17c76b682ed72e8388f020d6dbcd0b7289c95fb2a691c7,
c669950a39ce73eb5336356bb32376a2a150a25e1472e98880a80e1fe279e65d, fe80a18fa969879b5d8b0369d4d84f21ffaff024d4a20c572ab686b099df4b18,
b7149ce223d8134a556099812f5af8a3b75f55ca511b9c2e3a4674dc1ec36d8e, 4a1c5ae33cc3aeed744a51038df821f617c2da419f9f50fe9f709d28821ead03,
88c8201b80ae5cf7cffd71f99d723a0a67668bef2f6b1bc03246cadc2f749c65, e9c2bd1bc41d47b166f4020ccdda248263052c2ae4fdf09862ee7d7e8082ae31,
232022fa86dd20f310f096122e48472d9e8eeaeb1629a570133f436ca76d5edc, 66e2c24f55892790fdd042caf1f47bf0b16d85bae26693852e8730d6a8580ec3,
9a1fe7ba72f9a04c1c1e5e7310301cf7a111dc2da6902e5deb302d0e119c508d, 948bfa4ab9ca4794394ffb156e6802856a7ae36933f91dd29109d59ad8210699,
a3a9101b94fae560bb2b7d1ef3c8616bc7771a44ca9c8bfd018eda3fa550b0b3, 832bed4187cc470e03c2db4f7e42f5adb53d68c280879e701103b6bee3a0d8ba,
5709d451a6de4a070643721f4596b6bfd5380f26da7a27d1d915f93753aeaee4, 769c60e4f420f13aa2c3ee62337a73f74011d5eaf8892b3ce873561e9c5a24cd,
61de7e4f717e13d7fd2056c2e222a8e7b9e546babd528cfe6b2c1852972c15d6, 61f1c706b4aa72a9e64a25fd2a082181487969a373020c37a176800a1beb99fe,
499427f1bd3c085082ffc1c23faf95540c6006d7daa70463aec51bb042ac2e8b, b2605b21f6eff598c7b446a6cf4fe13ce74c6befdf2069aaaf859a967249cc38,
7859739908505860df5166acac3c2e9f5e44c1ea4258a0ccfc2fd5e9aa677e90, a5113a4b8746b7ffd7d0b6d2f9fa77714a367c891518bfd341499364e4b7c0cd,
8154022d610dacdea08c7419f7baa45b4b62b2fd0533ff0d5bb5d6822d028110, 5aa93a29dff4cd520e69c87e93ae32fb69aebe467227b5354442fc171cd92173,
463681c66ef6aca7ad1847b98fe3ec0ee3afbe649f8740b0a568180e7ad72d95, 4ba1d318683e2711d5d98aadb368676b93b9b161ede80b84f3b4e99cf823a706,
7a7971d366a136f81e7ab43787184ebafa3577a2c40c1df413c199f1d078253f, a0ec014e633b1a0def25d523ecf81e5530e091be1c9075fad7af8aee97c86038,
389c7dec20d3cad1a873ada95c9036ef8d6e882023d7a1d0bc3b962c76dcdd51, d82741954830656962418f043bbef70aa7e920499b638e4c35b66100f2c32d95,
c1e64bd45bb1f812564c4fd788045a1f266d807ba95da7c9cae1c991e0d367b7, 8a78d9c87011e54dd2fdf06c535b664544826510c8d8518b088bb992a36966ed,
9799204773c27134d929bdb7c4f026451703717bb7c25a8668727a7212ecfd29, cf43f249f89390dabd5bf53b906f9d5fe20c5575b0313997bd7109d61d8418eb,
5648004fc4788b2dee4a64151f85fbd7e9831b82b9336e1bbbeacec0aa466827, badfb2b6d143c886f10b71219e384689aebf932316cb8af0480e5f9ac32a05b5,
4c227ce66be3f17ea2c76b646263d943c5d1859254d717e97494c8647e1b1de1, dac4ff226287d965ca127ff25ffc792f8c29baaa31392e29a4fbbc48f22e4fea,
fbc1432e9c0a84e9e878452825c3acaf3efbf9ff1fe489c4398e5d0ca7309370, 76e970b5d30f6dcb89225c31d4004b3fc683ca58c86ac148b139a3638323e089,
175f81d73067caf0837d8a38a3e394bd74993e34e2658537b072814cd87785ac, 1d6239f165f9c40a6dd36054fe99768272bc62a2054fa1c30f5af8085b36d902,
a296ebfe5c73251000bb0769569951336ae7d1cb6f443a2cc9d63f111b8c94d2, 1d80a31879e87798cd9d578a8f510494bde6b7c43cdbc5ea326ce59f719ee327,
c8b7046749d78a748dadc144355b1a21fe1c75317dac2818e9fe777a53f66ee5, e4e5174c1f429d198825d8bb72262a02c4050fca8df8ec59b73b2fb18b196e89,
2205316c04a1043e670fa9a8ba7f1ff427b8c9b7e0e265bec0bf32a3456521d2, 62100bd3a9ef310b12154c0c69be50d37ac3fa28871aeb45a289502525f2477e,
464b61c19f409bc7035294b50df365ddab4c61d5c701c175cf56d023ff5e927f, 389902d32f67555f8bb9c959c3654e691c680d24e20b5b22698d2d9407e3c561,
c9b33c978ed9e3f65e71c451356226ed5fcc09480ddfa3b3c2872d9ef4a13a66, 3e56aa0a0a38f49a7f2e84bd49ba3112d757a84b3a15a36507676acec64c78df,
3328b30707e39aaedb06fcd228cef56dc8875b05458f9e8b9eb03388bf1f0af8, 5303bd1ef5c54b43190ada36b47b6f14baf0d04a5ad93a184d3b8bedc2aa6ab7,
3093cafc048cb12699f9064ec50ca98c22e7be4aeee563b796a07ed8549fa561, 20439fa4a6dcbdf1ec975cae5417499a448e997d8aca32ba809b0c79735e88c5,
f2e0c441a8f664aa7acd7acce35e861398a3ad63c208a4cd0ae9de056ae9645e, f35906d81b7e54a0c8b3c411a8b09f3e56a9d111a3999c9233b5fe61626e81af,
7a103c4ee48a9c2262388ee818303559f62967efad5c09862ea22fb913cd9d9e, 859ae9040b9925c9479466075679fda54f7a7a48195e019a5c13a44b9e23ab10,
c01010528305b60a0a3ead5e74416fc540aba880f7c8f8e9029d5b8a1a57f8ca, 919977e9bbe0c2b41e5e91b8e4fbfec8d4f22552d1e653a8a5cf038c01886349,
9936955360f4a06d638d18be5a8303486b82d27dd102a53e842d42a353ee079c, 3ceee2eb7bd089d0204f9d89a43e4de15d5f917bd9c8be87f46d15266e352a3b,
4400e2541abba700fd151bdf6aba01f1b5d9f6f80a5038acd6b37fbc9d2cbf59, 694d6cb8b81f030aaefb7f9c933b9bed8db177efb07c30e3dc8d937acdb8ae5c,
1fde6f3ac7f84b86a5a83ff968840cd1ee337c637e816c094bc3f006f539bde5, 4300a74faf8a1ea70c860855ce0a04def4dbc354f4e5940869bc17e6726dc821,
91ccce773160477e02eed7562e7ed9ef97c24a5938e030b328f4fee83bd789e7, 1afdc983261fff19101c2dfa54b551e274cf007a8ceac606f92ec509db11ac3c,
3933c8ad78a684ae2596d95ccfe987ed7c57b4ff8da016c9a1cde13842229234, 91d785e0ee4782e1a0a2016d8e0df5ec698ad4fef21913405857877e9eac9513,
b0d3a791ccf357479924d8a415910bf0cdcc7cfe1e89c3efbd8a743858854b7c, 7589736fb1edac7ff6018d1021592b3f2abaa8fac0313a18d5315046a268bef1,
92e59fbe8337edc0e55b35bda0bd95dd26da82503b4ce3cee80be570319c7fe2, 3ce537999fa73774ee8bdb181bf7d8e35b070b8aaa84094f80f2d30b601b3641,
6e57851e00491bdee27fd986c14a6bbc83584c775bba51f9dd59997ef853ccdc, 4cb502f81fba5f0bad91d2431d9f38b9bdf5b3f43ba5150547e23143038f88cb,
75fa06aa13657418fb8aeff47db7650b2f96943955cb5cea25974c2ed4a7c673, 9f4c9d990a2d2a126775a150d59cd7d05efe54b8a81fe07ce6dc61c0bbe40fb4,
253a6a66dc5a53f183baadc2580fa8cdb923d785669153a4d1f2deac8dea0c5c, 7a34e40c2e0a1b1290095677cdd8878b89309ecc3652b7f4c436688312d4d21a,
e73e1e2d86286c4fb51c57e4ee0f2c4d091174c3658170844cffaecec093ace4, b7f6e9d744798b8040c5ad6631dcb5c25c0a28123364aec8d3b68aeeaf392286,
fdfd7208a48a559959e705806e55784958e571dc97d4570b9a58b51ac0ac9dc7, 0ca75ac7f632954b258a5e4ed504eb2882f2ba707a81b5cd8ab567fbbae12e7e,
a2234d128d1c260e34ba7098aa15c68126d8e077c51f55bd16724b5867b3fd34, df2eda61d01519febfc1d86f1aeec9e1a684ef6df5c612bd79ab6d15e1520f95,
2a0d0ef7ddccdea78fd5ac94d2acce696fb3e2e3c7d4d66af59655baf29a73d6, 3771c601e10aaf060cf3f5e44fdc45e844950ecae8673469158b9737503c6967,
df3592ced9cb1b36953cb7d806fa56d3f580ea2d754c654ed67bdcd943173d4a, 76fd2604cb693d7ac93aaa55dab446bbc2e443a99d7006de03a0b2b9181f46ac,
69319bc8a6f296465e4981288104ec64954ba6c5b4c1b01159b4981eed559cca, a3d0f659a86bde22e3fc4bfb1db60637e2de715b63407ea4b49a61a0ed8bf83d,
6b6e04370ace1fd74ee88404acf3069285aed0d3d6d1a478f35af70c84e6eb1e, 9f57aedd77a352c89065dc5166e60670db7cf681d6fff7cbfa97d074d8d71762,
dc5767eb6ac4d8c31d2cb73db3b2237f08e776a35b84dfcedaef3310cc7bd659, 2af98ac89b277ba2a26e4211a751219407eb4a39a0fa07b8fa7d2f672eead7e9,
73fb069209933b5ac865869da56fd1ac55cb5145df3cf5bf27957cf7a9431037, 5e297f087b890f132ab5c1e59c7c278cbe18e54044a5ed57910d2812f55019eb,
3cc8e22493763c3f2c53609818fae023b6070f0cb7128ad429e387f836cddba0, 0ea9a33bdea21236ce91ca0e223f51b4abcba9489ea35c4c8b68775a18959b9f,
5a11c684f9cb6e59aeaa25400fe5e334c6a21763813e2e016d7eae458169cfd3, 0861826e6fefb273d6423b5000d89843ee8f9ddaa173d0c3010fd2e864c767ef,
d17413aaf5f15ddc0cc42a43b08735cc9329f89d34925f3b7ca54737b2fbde18, afd77edf2db2f99dd2220b1cc2b64ddfda40f3c23c39112194e7b60a28736a3f,
577acde222af8754bfc12c3524fe080225ea1dbc2d3a5b5dba3e1659634ab74c, 19dd32277da69516c2f737372c552d101b4779742eb0add8e41df9d20bfdb815,
d1d966589bb0f3b51e6fe9e5b3992ba4d898d65b3390e3d07ebe54bc17f1f66b, 541cb9b45045168255c2ed611cfdc84765e64408d94238b29dc4dc2f64f51399,
e8f16593db1aefaecb1442f0fa410314076a8dc7c96f79c54111da1e476cc543, fc544835cdf7b1a11bc09a039ae7a43ea827c6d9dd7803f8112133f598941291,
ba4808d35396a7b11a3f49cac3b962f19a49ea151c44812fd1fd87b919a4c292, 0b940f64f775f70fdbcf68c16d1d611326948810fb7d308dc1eb0e08ac5c43cf,
a1a420b2b6b111197f00b31df0124d1f41ec287a4c11b4d4c25daade317f1f12, 7a46c5034d3aa676c3c5b98bf4a16bf5485e2042baa1ed8a5e8bb1d441017d85,
6cbc86bb20329a7242c091f5f4e6c0f338943a70d5735da2542e6a5a8411fe81, 3d474490e8a40a18d98d0ebe0b8cae12b9379e540c143ce11c5915db3bbb4741,
21de45facf782cb275dcf8dd322d9b982e75175caedd4d4edd2816e98892053a, 7fc7077f4345bafeabf9f8bfb72ee5081501495406366b715293929585942f12,
0409e2129a06f9ab0ff56dfc68abc615b495404dca4d6139f99f0c3fa5e14235, 46476758d53247c85f91e6da6f46ced1952e732174a2f38a1cf8a14eac7c0de5,
18182ad640313423ccb964a5061a17b49afff826c2d2b066cb3a78415f5fd0e5, 5be5d3bd22607e0a33f71a3b467aeb783c416de5b20908a124879aacbe8846d0,
1686c8f90e5de0e5333ef07f8d3db895dbccb1ba1fcb4ed1f7e3e71a11c6db9e, baf0d35c2cce3f3db23917865f1b409d4a3b854b3efa958f6d11f83bc9a98324,
14148a33a5c980cce640b1740b5d73cf3532eb440647f680f2e4710d07197ad2, fa7b297e186de0e428b0920e106073f9879b2173e55bfab3086ed99d3527ee2b,
06882978b1de2f4f67b4808acdf06530db69312bceb4775083b08f379dab137a, f14ba60f23f9a1cbcaf7f03fb544be35fe1c3c5d77cf290f8c913da8015f1a90,
585467a011cd85988790646b5591ec11ccfee0430c6ba24d1d0db3813bebfa65, 616bb2a98f475d91f67015403be093c534677280fa0c30f8cb0a3a9a14f172c5,
d49f63b3deed8e1d05c9a74a36a0d417d97491b5c8a756e12e57305946221681, 60a5c39e4d83a3839a81fd2c7b00baafbe11f46be0330cee75c79616b8984b30,
d8ae2d136e4a1c521a82ca793b3cb650e9ea5461a28e8553632b286f13d99e8b, 38813ef36a0336e1a0207eeec5e3c3d0bbfc31468af432de7621c8f44e2805e5,
76de78d09d9ebefee65e4075ba35f4d6260436633c384a18b891db4c49b35c03, 5d5cc7b4473ac27b5097e02adcce41b79b9f61e55fb7b38c039cb1deecac698f,
318fcaed1f9f2c206d586f1902c65777f65410246f6b1450deadda6f007e6f22, 3cdab8bf8435981eaeeaa5abcb81a75e02e92483517ea995d59ec2b21b636885,
10814460d1a2c39ceb6361d95b53f18b1ca529511a0a9df14da11d5346ff5723, 43904a7d40d6e61241c10c6cb70a6cc99b80294238122c32154b44d818044851,
1b7b1d273212f855a3714046167b0bec5e635c6b7c78089de43d14dd1d7acea5, 2fc9da1ee4d34ef059605df1a095006508d3885b9e8296771a89d6179542f431,
365a541a989681b9acf920c7f395188cb3f811b4d41e1d624aa45a65a742bc51, 7c5927d48e4f532754c8422d70b55339b0aa144aa6aa645a3cd341c53aeff29a,
0ebc1d7d0eeabcd709cc7d3fd0f548c6ef002093c40c7cad877159c10ef0d74c, de00e51a6c0391b073312b4c71ea03dfb6388b7899142b93bc42e7323c30deb8,
ae14029ac5c9bdf5e9b9356095278823f2f473fdfe13a24311b12649f9a7d285, 420cb3ca4224c0dce8a55ebba09571f03bc7c0936232becb52e33a20be386ada,
bf372545847f4f7414873a7e95e286cf5ca5463d45627519f1b7c9cc25692bcb, 75d9efb9ed1d641a2a06300f048906b77766b50ee54e3cce5eab9317d3bffd88,

fd18f304f2134603b8434e9acf3f7fb85782b4bf8dadfd23d0146473b1bc7416, dcaf43b86c33d8b2d89d105a6ccea54f3ae4c41ef1fdf24425f1d1f067a0c3b8,
b29fe5976379050e2536753ceb97715a75d405e91b110a1de9165c4ccf70e6ce, ae6b0b91e9edc415424513ffd48b3296566f31e094f74fa2a97f0273aff59fb8,
7fe39651ea180636f7de1d625a6af763fee7b955c2b60273093ad17d7757685c, b15b784924c4bfe305752a6007aee6c2a78b741ca6a859195765d7576fedf844,
4fa7556d71d2a193fb39d74466c1e7c5ca9c011d9ed26e0c1b526fc227629d71, dabb81ae7ef8c0215d0f9c0e2a3cd93170a42dd9de5f41cac77b49408b96c1e5,
e1d341a62a1e93877ef77f8a4cd7d1adfbcefcb22235784a08cac5d373627b5d, b03450762d7f7897a6f0dbd61c05f5d3d7feb7fc236a88fdddad1f0f7aaa0b2b,
116ab52a3f055f8f7ded34ec072596286b33c0bcd8b3ccb96ec9cc35f6809885, f4c483f9c1838080f64b918aaf0304ece09d41d2dad0cdce769b8927e0a9b5d3,
cd1ee57d9a6b79f2c7469b511ef534eccaafaf89ef4fbddb7e285524028ce859, c4a511171b94cf71f7a363da41ec6798b14416132f7199c0e1fb5a742c1fc08b,
0692f686d90d2845d7e93e5972d701d9f50dc9a70b1f8ede1249ed82aca03d5c, e0aa98bad59e2943ae6faa5526bd2db1b3ecafbdf4a72bc1a8865b07c05ecd5b,
cf1cfce83e2e7b11aef6dbceb0ec9d90cb5907ac25762b7c89ab75e30854f111, 52f83397b98f77242bef64011fdfa4e1ebb7275176470563c9d787f0c3d43f95,
1c8c91b57b99bcfc6bcb548e80f772f262d2cd4ef4db46543c3e65efbf6f9d9f, f3a027ae48ccf6573d4209d3287a95802815483f70f56dea01f33f1493eea711,
ec875a644ad8b2af6ca9a36936e175b2d79b29ddbe74d1ff9b48b9d8f1fb793e, e57ec7eb68bc259a0f8c44851f33b8ecb7c84171dab7b8d75e1e1d50afe1d9b0,
4ebe40997d1f3dc709477dc224b6d3c7b429d2347c678b55bd6bb99a0762e3b8, fb564e7aa0e42e5332742519df74e722d6d4101927ffc4d1a173591cc1f9f3dd,
ac2515ace3c5f030d1fe7eb2658dde67ebeb4248454803e7fe0ae989412df8a9, bd0a194efb45e78e48af1cb80a1a759202ea3783f2b2c92fe98ef8428a69e349,
2f9f54cb8df35b01cc2f9180fa30e1d0f8f6af5043d8d323ebc234e91a6d53aa, 963baba0bcb33c21f2a3f839307cdc7f97770b23d14c56d7cdf2c51cfeb6fede,
654f0f7ae4f6649227296fd5a5febabe49110a2393e16107f8ac3204d051386a, 0e9244794279cf209149510f367f2b3fb38bf639bda31b33b9351e6429a2f1c8,
449991cc9b37e9627d0433ecb98a1c2dc7130a815bc4a59b0a0457f924a8b068, bd265c4f8c50e8fcf3cd979f43ea52bfb054d9ac3adabd42a13bd85a8d60c779,
a2642e17a50eab9cd2991360f8fa56dfc46ace7e4e1b4d011aeaeec5696cdb88, 769c797e66215b304237589ab0549ce0d996c80bc518a9a576f958ab3235b2e1,
212741b658d5e5d14f8fb9592a7e699c2a308240e358d64850ca0879f0e201a8, 73cce3a6855f58ed8f2ce6d7a3928705c96eb96622a33239898cd1d585485adb,
e3626790faa9d6e76648ceddc80409acaac61c9df908555e53ada6e0ff11dea4, 7aa67e055f61aaa32b75e28a7267fe71388bfaa50488a48ed4722177987f7cb8,
7a3c59090078f01f01c11ae9d8bdb4defefba1be2adac3a6c7e7d8161a2bb742, 2ccf194a3cd7aa45c7914085eaee0ce76ee9c75258fd5374a519235af265bd8d,
22dd220fab7aed2d7026bd5ba6c41f9fa11712ee0597aac62fecaaf5985650b6, 4e1a1dd2fcd78b5f22cd234d4e9049a493dcfe3747060e2be27ecc759b971bb4,
1c0c63022313fd7cc40d862a5061f9ebdc712ab6ff671ff9fc15c104bb3842ca, afcfabc26b499f32ed7b51636446cc8999bab0a155fab59c50c0b4d2f9125cab,
99f3956f39bdeff1ddac2b12129e3e689c1eb077e9f1d1389c7f43100919fdff, ebf9a44bd59b2707957f377255f527e87775bd33b37c4a6460edf6719ed0b8ec,
707b1c4d9795250bea4c85692cf2e3ee9bad4d29493653414820522674d4bebc, bf193894bca2c6c66921a3086b29cb9a253173066445b6026a20f7d8540f674d,
58ac3266c78db9af325e48a8015e40dac4dcb2d9cac979870ab6f8faac63a591, bc235eec1eb17463ab36b696b423dbe57fb5dbcc8c8c653c19915d43401c6026,
74c949dee2dc772fd76869a13d0aeb6fe0c154c9e840d25ea34465cf72c84ab9, b7bbe1d3a057cd7dc4cc404f0be69775b95835b9d75b5efa51dffc3924b6f69a,
e32f9fc06c501b0a18f25b43f20189e1bde9b85efa7e3e349f59da41fba8ab5a, 4e2ddaa0011103c36529e4f9d0ff01ad2fb8681dd810baf84adac9fc6774bdb7,
70f3878b7c1a8522754bd0ea4bd27f4a51b3b85ac61b8f123e561f568bed5e45, 87a3bc58c087752b57f018b49abe1a9d048555ef4f5de2874be0ada2cac2d202,
730f0a7a9927d0266cf8d212ce8a9b41cb204466459d0081de829ccea582745d, 1176e9eb27e5d68d0b9ec0341a554c4f62b54f75c5223407a29cea2bb44db58f,
196276e18f94d74df7b761c7a536b47270223c57369c7d389630aded28b2177b, 3da5bfb2b83381ea9a4452c83f9a06c52df3b5302c87b4a9625252d9e732a080,
4e3613290554c39db23a931a9204ddbd10b5f6872fbbccb509d8fd1481dc479a, 82b73d721e253eb4b7003b4dbd1af770009aec9f3033ba949c05e63550c1d64c,
505cca8cccc352889e17e2b8db7f0cc80ae918b34d808f2f96e9fbe935930be5, 04234b843b1402f8d67bec0970487531d1e8bc494ab47d47f93e26881ad31372,
8db8bca37b9ac2d69e5671de8f8964078bea49a480f86c4a3240fdf98d2d048f, 1fd2bf8a9e2251699ff14d7a060bc89976cf9eb86c2a8706654285921f08dc21,
c22005efbe4fe85ec7fc420abafc94fe375138780e40d439647ae23d0560f8ca, a2faa46fdd5949c8141b5254a7ed22476cbfe15dfb8c7f043f3f844346ae1b98,
c41a55f11f18f37f07b237b12df45112611bbf0986002fca07e7dd41d6abee25, bbc7dc45c87ebfdcdf856578454de8cfef6de85dd0688fece8ace57e15ed5d9b,
09fd90209e45f07217ca170e4868bcd919188004ab8ad10224946f084e3cfeea, 13b5278321eadec89bb09d9ffca795039c7f65211306d60c30d65e43cefcd7ab,
3fdc79b5058de9a1ad100dd6b4950e164615465019866ff439f540ccb7382625, 016a95fe5b53ced507d6ab9ad0bcfa9dbcfcaab7f0cbf519883efd5aa0c4c0ac,
c5b13f4b0f176366a20538e147328c42e18cded593f6aa877dfcba970852ecfe, 96f58098068e582a581c199ad2771591c8346bff6fc8b341ccab63317c2b1b65,
41523ef717e66e740375cfd41b059a2af330e0aea7851bb62c5b0469917d76cd, 13925c05482f45fe2a7de1d55389be3097dfd21ab5f959f7af55c2f5228b4a07,
b8ba02a7efcb294b0ca570c52e5102281a6ce935b567ec1031d076110d5d0c58, 0883a3ae06b93ffdabc089b0e02cd653be5b4c6b0a65bc6e7e90614ef79ac3b9,
acafd63b299dae636e114c26f1db4720b9c2d933475d6284782cf83c30e18a87

A.1.3 Popular Apps Collected from Baidu Store in 2021

Table A.3 shows hashes of 226 apps used in Section 3.5.5.

TABLE A.3: Hashes of 226 apps collected from Baidu Store in 2021.

952eecfeff057f61558ed46d1d1ae5a6dfafd99c144f4b3130b691a7b9c0267a, b1097b786e8f925e38b90fa57061e69a2b946cf8159598b2096b767ca162c1b9,
17b5e7ab76fd77dcd68bce9c3b38683c7054e3444270e714afd30505f82516d4, 60727583716e1357e7b884a2692649198fd5a3246ca6ca3103a8b791b9bce515,
d551edb189c8c4806342320aaf3a930dd809684667a0c151c8ade93ebd6d4a91, da6551950a3616ce2c203726a9100ce7cfcacf9eca23b25e3c370b9a456165fc,
2116a434f2366435eb6fced09974451e1b8d34fd71b7e058c4c21ad2d9dde179, 97af283e983966d3c5b79829c8f5fa8734673058bfffd750852d7455c91a3354,
3f54240f78b136ebf40e224aa6ed2b5e186c331c6ff61e205fa2118aca138eea, f4cb8268010810ce7ae397a48fbbd7d1c858c87cc4e8225197569436dbe3e34d,
ec13d78d58989bbd8c93cdb9db1d70d84ed48f61f533e6a7c2308481f701d6d8, a50323f345790b62cf7383cbccd5b6ce045243d29d890a450a25afdb1da766c6,
f57caab87014c3074d156cb7d1dc6fcb7689f8b59d793f16a7aff7b4c75316d4, 94aaa7b75e633da2c80c8a4e0b8b37ef0dcf8bedfe71897819d5541e94f16e09,
e773a08b17010aee8d23bd181a7b19a98a6d6585be7be8b0c2c906f8e74252cc, a9d207c1ebb37f25591dfdcff43174405ec0000c5870f08c2db9359b1adc42f3,
653ceb349499e900b19d17d50e09a166513b806c7ba85a454691b5d28d8ddefa, df758b0faf2eeefd84499aeb7369e0eb799cb96d69d9320559a65b722139d97e,
952afcaca1b38c968d63a06ccec83f741167480e3d1e5a9f5bbd81075a6a65b6, c7c9cd5b4f3d7b495bd154042103d6a97d518b27666f1129159abc5ae821366f,
618561f3140efc658b79e9901158dd0efcaee534412d9eb0760c42f25c9564d1, 1077799f517c50a7971de86c52f3dad001103214551097d7ea04bbb95a035562,
94b524ebf7184be60c921a2243fa707790af767b2970e0450f3efb029e68eac6, 8504cfd0addb20b802ea1de32651e260e750b54613e04ba377b311140e508793,
2fa3f29c44d880e4cde54a8a5c5a200bf5b830c56298f62a3a84e2a2c28913c3, 81936b753c766bc6c8d4db9aedbc5ce461bf8fdcf920305c26a4bf5fb23e99ae,
b51986f88822dddb74387c987573e8e3a1dd214258fb4352c1d253434b27e580, f5f630ee3a6919e11ea176ab425dacb24e37ddce8b12b56dbbaadd6f83ba2cb8,
17c0ae61e915aea6dbd083ce5c40b0e6dec8f228407ea44473bf659321ccb8f0, c5bd0a569845af73db3f145efb5bef3fe92d78aefa24fb05fa0d1badfb0a8cf8,
7586114a7da3cdfb699a0e7dfa22f57b7cf239258fc54e6b963999969f75a397, a606bee172dd881b51b14f7dab8947509d7a437771f7704f49035f4030430404,
1962164850c9d37211949f69d8d3aa167dd7765d4255de2e6364c725f3fe5f51, 94aaa7b75e633da2c80c8a4e0b8b37ef0dcf8bedfe71897819d5541e94f16e09,
0ab42c87f2302cc14b36f8d0f0732ef11bf6e4f25461c2317cd8a05b995bb202, 0ef73c18c10ccce1c61c8913275262efc359d535f773e066c52bb7652b7dba6e,
63be3b8406815a67c276015dbfcbb04350ebbb37998c07edb2904162dd603c57, c92fb2a73f0a7f2db116e5bad4329c660ea56878a55a251cb6ba532389c6d02a,
57d20578315bb93439991e0ec6b127924f07dbb09f0c51f373b3f93d1a5d57a0, 2fa3f29c44d880e4cde54a8a5c5a200bf5b830c56298f62a3a84e2a2c28913c3,
c546f9baeef431a179301f93915c455e893a928ad695909ac015b4b7c2151334, 2fb5504327dd43dcb23fafdea1d634c9daf986d7aec0b1e49882c38211fc9f5f,
6696d05f6e59f47d280b6abe26139ebc248424f3b47f70c8c18e296f58175378, 9127dc3a468e8ac1ee694d52fe3783ba6ea25a47be865f04404f965cf9ad0ab3,
821a95ce6439d6b51240ffee134619bb73728877fb2ac915cb79511cd884b07d, 2cdb19167f52f63716b8eab1b894850134dd6f3373fe5da4903bfb7292d214e4,
cc427982bead017bb31846b2811af11c6b06d90fd7c9094562b2d9c23045e0e1, af03e908784fa7dc51fa6a3d0d7a16adf358e53ece2764e6983c993ee7c5d284,
cf160460d51ec84092589fd0bee97b1fba60acdb665c042db16287a9e41f8137, 9fa4cbdf40174e0738c4bcae89d8eb41fcf9b7bb89067a04c104f7e4e2e64916,
808472e7ba33df781df0134af81bb70acfa6f05a713dd23986d9848c5aa417e2, eafd48c8b6b1b42d57b71de8dc7b37a4863002812f403d93bf067e5a912c123a,
00d6f04b9b508608d9d54843a6711a340326606aeaa9cb81039e850a6d14a786, 2ec6cf031fe7b7633e375d364c55317ac9ca57f8e95746e5e88a963e4bedbc38,
13a51abd33521017e0b284a80a5f34238d6ea6b4ff1d98b2cd18a2252fbb1317, d6a414045b75209ef10b01d85277170d9620dcc29264e2450f103e0fd83e8bc6,

910b0f7a8cd2caa7f2786549d889478bed38b25eff50f3596297ab8c2faeb9a4, ad101f0bd1e541897c0bd8bb49321a738526995abd3cdaa1252613763c00f696,
e473ce43721c5562d2796c6761dd355bd070bea6a06725d233b2d3344ddf22c9, 3612b2674a1878f0424a7649ae6f94620af4d6b44fa3e62c467a453b23432bdd,
8930aab838b74bc5308ae3d6fd4086049be16380416fb1b40524fe5dea653806, af03e908784fa7dc51fa6a3d0d7a16adf358e53ece2764e6983c993ee7c5d284,
6089f91b5e8f7e0c506b93b3ede270bd935a3c890ca04730b98000dff2ed507a, dac859e4b7dbaedebe20b8ddf97f465fc0daafacd0cec35392a78d7343c9bd1c,
8a8260c9ae107c7d55f03f8d9848e33aa88371770d42f064f7ed88137315088a, ea0c1c27717d37cbc584e4106baccf9f901148db81eae89bf6c87f452ac8e52c,
10a2663a8dd991b0caa7809cd206fac6166ad4355c2354578f7ab46bed380fe8, 380805419d1406d7c7bb2c4613fa1ee0515e86e84dd005c1022eb0f34c2b4784,
11f304986e967fca41ddca87018ee4322aed89c899a59fa990f0b68dbb22671d, f292dbc9f10ac7cc473fde545468ab626907a6672390c8b0c4a7d85d2d671d52,
e094d3d9440d371e39da62eaaa41f18cb9543247e82d1e9af327c4a6bbe01e90, 75e0e8342eaf78d5bd9f846562b621ee9e8b56dab02ff69f650364066e55e3c5,
5b3d2b38d894d2a1e8f795f4886a01039f1294da539fc6470a6e80f62842916d, c7aa9acf7918dd882ce22f8c7079e2032e1e2aba5afbe2bce65ed425362ad9ad,
a722d72490950ddbc15e18eff71e60d35270ace0d0f9cd1a422f92be84777689, 9ef3afe6aa9a31b08d9343d3dddec56f8856703e9d0fef658c6997c9d955671b,
761363bf1bcda12f908e1eac1e7f122ac102b0aebed773c4f1856c9cfcafe6d4, 60f285660dbf57636631d4163983fd1cbfbfb5fd39b7c27ae537c9c5c29deeb5,
8164305283bec40010c0d4289e0d3af62fd0fa5b2dd1bc440d5bc0141618cf61, 618561f3140efc658b79e9901158dd0efcaee534412d9eb0760c42f25c9564d1,
3da7e02364e0f5a6704e690a8d5a563b492c366b0ba49f4e104a62304a127a6a, 2ae85b554f00d3179c5127d282a3042e4a8af6dd0938f41417802da7fe952731,
8504cfd0addb20b802ea1de32651e260e750b54613e04ba377b311140e508793, d29f2b3c5b6afe5fce8c11d4cc04526f4a2c1dda7f34b3b51920923dbd587fe3,
2f2b0d19684a0fc4139e1aab08c6d48c83361e8b71a0742e9dc7603c3eec76e4, 8e979e63a70cd87f23ae91429b0f27550f08fe95d1dc4b191d04cd1513b23057,
2e7c471f3da635f8a8bfbc761ff2b6606ac2070cf608477e9a1ed3bb6e76e4ec, 0ec71d2b1a806a952e10dec6036263148a0c61eb3053d25c320b1b7e720eab97,
82398374a7471b7ed255b24c64697847bef076e229044203d09d24b2b8a733a6, c076c2df92638ceedcdf5e6e94d9c5a031a90292480d3ae772e5e085a2037345,
dfb193cd473c12d8e5ff68b7b6d3189a361ce9b9436d46518bff2806fe7d4232, 290cd67aa08e79982d296e196bc853e955527ddd22e1ef8410d1d17b58307502,
25df17f42ffa96c64f97c02d157ba587d294bb0de4aa90b2ef594b58aade2d28, 853a4ed8c6a9fa410b4d115916941f591980a6e0b8bbf06a5bcd9d87e3cc84dd,
c546f9baeef431a179301f93915c455e893a928ad695909ac015b4b7c2151334, a57659fe9344836a2467addb01fde979d7f6f5aef4e7e4ab501ed91f8c3581ec,
200e1f33340dd29f8a1327f3af76a6b16f2266bc3f4da641e36745d8ea839bac, 3711ddb5b71ef874a6fc708549a14177562797ae8161914beb772462da20e318,
8856ae294ec4a1295ac1ee16ac0f23aa067ae0cfb212a802a55d34fcfad56c17, e473ce43721c5562d2796c6761dd355bd070bea6a06725d233b2d3344ddf22c9,
82398374a7471b7ed255b24c64697847bef076e229044203d09d24b2b8a733a6, 5550df0efb820578e0ee649dd99af944b6676e2948e6504c2c1f6d4e83a2c493,
32ef18172b8f056299ffc4e72f42a09cb59507451a9a84ecaa7700e6406882b6, e5e50df260bfaccd0e1079429fd3e28a459c94413f43cfcacaf10c111e75ba1f,
988e74e669d85cc11247653afd3c962c120daf9be1007aa58ab073f544e86a90, 7fc1ab5a6f6dd068b940c3dd74cef96eeb74c7af8a16277540fb64e41f3ffd91,
0ef73c18c10ccce1c61c8913275262efc359d535f773e066c52bb7652b7dba6e, 28ef7765a314d46a6294aaf5c584be57b4282c0faef8086e71e1f34202aaa43b,
8759d816ca8333683dfff9b643df13edcce9f821f8bcd254ed06b96ef29ebc7c, be7d34df81a8ae56751f3554dc43dbc2780b8cf3b749728088719de4078c35b8,
cf55e68d71cb610a8062c3dd5b58bbd5597b5e942f9d4bd1162f5f1dad9f76ed, 4bace749c19ebcc9fc431c35cff36bdb33d3615fd43b3f866a830512cc0063a4,
ce1a09dfdee1fb40b86b3302b9cd0febb4664d0f8286144bac05dacde0471722, 07cb754c25880e675a46b6accb0057322589c8afcc04a0c616a49b7a4d891be0,
861977515d6fb8392eb330e9922fa26059d499d52265c6dbbfecd5b68d3afdd7, c9adcddeca0c635102087a653b5617d943f40ebbaf9dc2633ec012de8e737c7c,
3e15f4b7eb3d1b012b539fabdf612a5acb3047f9547ed5170c834f4bcc3166dd, fff6282f06e51083e826ba61354acf728740ef834536ba860e8ab1ac62f97b17,
aa9ab040390245aaaa44c8a44d4ba568d29a9fc0d20123a13119cc51593fee1e, ca70c326d656a28e8fe84a0384cc2c2dbe4de0684a90fe497fc070579ab6129d,
184f58af1264dfd95c25e9da33d34556be4a23324ed2461ef3b3cee9324d4e03, d234993129d0b2fd918bcde0f3b582e197aae3e4ede833a519a811a696cba098,
ef9232d3c3fc8d5f11de3899233328d55cadbad78c4c58e6d8b6e43cb7e9597e, c83411569ab299d818a3f754351a6bec62785aaadd10581e3c3716a15aebf09f,
83a5d84f3de0a06905bd28602e052db31b3465fdc1019a5e7a9c07fd57b3eb1f, 5b296f098c3cea1b34c389c32df26b94ba31cad45cc7a506a968c3be0f1a34b0,
2ff32cc3e81cdf21684bba3e696b0cedb2bd05c831c5f8ffe36dc255f9bdd492, dff6fd5b4b70a19f15ac57e30cc4ceb7e3b450979d006d6b5b1af69a0f549ba5,
36b62de99ac55498d5fc48349cde6c966d806b53d03fdf27844a6e27a0725dea, 5159b131d956946b9e65b471156404c0722e85b0f93c0cc233226692b5ee0602,
01cc7e88cfa05a48ae430499f201fddb8eb1b0b02596ccda789b9bf664b16604, 80c8cdcd621c4c7792336588b38fdec7c98de11a1fd83113bca8122388a5a321,
f5e851ae738c1fdf365fbfcd8a4759e263804b7546a1193f7e71a3eba2697946, dfb9244579a695a13cc1509af81208b5833c85f9dbb6b20ab7f12fbe7bc81470,
b8da9188581982423b4e8e82787703d99f4e8cf0f987a175121dc6f9460861ea, 061ea82fc3baf2c38c2beaa0fdf0ad03280818fba7c0e7e624db386e2798b163,
ed87c28a78d4385a5fc4274f82fc0435fb8e6ff4e5791df9efb2128d37f86899, 7414a9e35c610f9012c128d20dedda2f852960e44df367e599e5edf247ecc0af,
14d7062cb09de35c8cd1830242247f03bb821c7f8d6467c10bb834e1da9975ff, 32aaa27bb69a80435195659832f140f91ce8b3a08a9d7a6f63422cd6e85265e5,
6757e3ae296578c1ea733b607d50af05b8e9694b7061ba43230a914b3a59cd86, 0a136575db49a45accb3fe546d568adfa5d9578ffa3617b87a109ce2f6c7cff4,
4f526cf1fecd4e051bce4edb3e5cefd3c4528cb567a687c398dc2ad76542b6cc, a50323f345790b62cf7383cbccd5b6ce045243d29d890a450a25afdb1da766c6,
3368b6668466b66921c5cc79b82312dcab2aea958a67680586e5b8e51feb1b31, 03f9ebc2a77dee3c4a7fe8db282d1f2743c9d1472cc7e01c1b05a421703ba8c6,
859642a7be6524eb420470cc677b5620b8b45cc07cce8b8d43995728452b7366, e00af539be94d99bee287319a8b1a6d9546a41ffa28f2082ce3f649b36882358,
d6a414045b75209ef10b01d85277170d9620dcc29264e2450f103e0fd83e8bc6, 2d39cab5f2aa58c18616cdba8e333b1c582415cd54849ca3d00933a69d164025,
7586114a7da3cdfb699a0e7dfa22f57b7cf239258fc54e6b963999969f75a397, f08f3d945ef1c8176f67fb133ae5ee314b943fcdcc6b7c17ac5e197022c1ba0b,
97af283e983966d3c5b79829c8f5fa8734673058bfffd750852d7455c91a3354, 46ea260d457b2ec76b5789f1c4397934c548cabcd1d1aa2ff5bd6ca6aa006c31,
5d41e81f64027d68372d4aa930b178d9be138b5b98fa0a7e73e86306beef06fd, d80bd7fb8eb50e4bb2e0a84efb506bf6ecbb80eef8df70ccf73e94363756ec78,
3ef46fc7e4f6df29bc5a9cf7ce1d0a9e00737758da9254026c72ec2e693a8dc6, 60727583716e1357e7b884a2692649198fd5a3246ca6ca3103a8b791b9bce515,
69cff743a8e97b576eea9d89bec87b88f8f0050f11ab2d9d1679a1966801eac2, bd9cb2e77651abb6bb8548d7dcd3fa60dd33ffa3b24f6de290e2686ab7ed6aec,
2ff32cc3e81cdf21684bba3e696b0cedb2bd05c831c5f8ffe36dc255f9bdd492, e8b4d1190129341eb85558baf46ca0866c9d9cd8e7111d690ded8449eb93e06c,
5b3d2b38d894d2a1e8f795f4886a01039f1294da539fc6470a6e80f62842916d, eef81197e631a2d18a98789f764c915f734f8383cc202cf639cadd9b50503201,
39fe53e5f241a1d5306c324099fb12feab543fb01c1eb7dc66375f03814eb346, a0bc479670334f74f72bc7d9a6c221fe3ae163d2cd163bf39e973b6aa12a6be3,
83677832db94681a03b8a457b06eb5e09dd4f1d1bbfa946bc8bfa27fd525f561, ff45a46f24e80bc36c488dcfe40aeccc270bdb10bd23404cb9c70ae0adc6f34a,
6f8060612bf39cb7851d0b2d1dbc2f8e33fef8739c70c6e3835762e38f4ef2f8, 24a3142e51d887939305546183137e60a41ab64fd99db66b2f35bc406bb7203c,
2a052d1bd62668f3dd9ee8f6500d8ece67e7ea3196990747b10eb627ba9ea475, 8edf7f731909eb0e26426fca6fa8844ccd1dd4940ab72baa1b7b9dbc973582dd,
aa9ab040390245aaaa44c8a44d4ba568d29a9fc0d20123a13119cc51593fee1e, 96a3809ebffb1e5986c8917bb9f81680012fb51900f1a589c74ec794e2ac3399,
256b7f314fd572f0fdf92c8b3fd75a09581d4142ced6fe583fac9d9fa53515b7, 946a43746545720816bb842304148613ab8436b53ad14befbd052dc72b7fdf8e,
868120a6b536f770b924dd297f9be6af1d6f65c5674a4bed697084e1167c7652, 3d54d8cd63c8d344d38d13c5fb0a1f171a25596c34cce080e58014b43f75a079,
0798c2cc8b8a103d3b71acf1421561bc370637ea0b5a623f18f02cc8bdc27b7d, 80c8cdcd621c4c7792336588b38fdec7c98de11a1fd83113bca8122388a5a321,
cffe11bbd2b90a24d562bcf0692b498b41d9e873e59f03eb1e194c4f5e7fa8b5, 88413ce715ea08967d23f06ba3cf82fb47f34b485fca04040db672a2fee5b7f6,
b5da24ecf85db79cab06f1b61f08c1e9d56e0773b768876fc9cad2b469facd56, 5b2db016bb8e087bae77152d29e6e94af2241559bfa656f5839934add3e6a6bb,
933e128585b3fcdb9d0b1d16932389158df346da043f489deaa5f9b15d176b6f, 458829446adb90b9da151f22003458920d57a3f767c3bbf87f093758e1c32c4f,
40399cd6a022c5a63148bf87e2157144e5a02261a6d0bf3fa97690e0ef5af184, f010046aa6d938853c1f14a5092fed50195e31fb5819eb7e6c88a3ffd6dc9989,
1ee84f332694983dd7a034e973c7a15363eea00153db13335df7e6e7def85384, 5b86e9290284ba6cd00741f3a231f8196b8553a225eb030e711ee8f9a1f23e17,
18e83ecc14cae83d8d1b7802b88b404326fe590da0f3786a8cdca8c993e9a077, 6eaa89e8b79610d5a0bea9e9aaa03a03c0625ed4381c7fbe46d40982f0fc75d7,
2a052d1bd62668f3dd9ee8f6500d8ece67e7ea3196990747b10eb627ba9ea475, f7ec5a2e28b938e54abfacd8b1baeeaf39f2b9cd0f49819ceb4a24a39f327cc2,
6345f78a2aee9bbc23904ce0405c38112d14d148f8b73397ed0a7c50ed5adf93, bd9cb2e77651abb6bb8548d7dcd3fa60dd33ffa3b24f6de290e2686ab7ed6aec,
4101ac99a9f5a2efc904e314640ce25043ca60f227985036f69e3072e769716d, 092d550428f8497aa475ec5fdea54c42d83298ac1c9a6b1ee035094eadb9b260,
6e6f2ddb7ae0da9471e8d5eee17d06c38c12df82005cf6fe92d495cbbca4387f, ad37a58995d7bd8d8727765df1b9d644472bc1ea8696337c3e29a48e5fc9f71e,
39f66b42788353a6d2832d29a2efea27d0ddea3ce7e15f462409e5d969fe8af2, c46488181ab9a92fc4c584984ee4b304a00a1fb827ac2e9ae0228232e8c33912,
304e16f8d87374dc8e8b280ec533c6efa412584727efee7828f45bb3cf1a84c8, 83841338e8ff48f10fa733b3118daacb5161a95757f5dfca3affae43acb963ea,
7a00e6b2b16e3f1bea8c832a2c72b80b6f7be8bd24b0055337ff33a534a9e29c, 48dc954dc40acfb8b70953bca1105ad111c270381d7e482edb86f1c3b3a480d6,
00d6f04b9b508608d9d54843a6711a340326606aeaa9cb81039e850a6d14a786, 3c6fb66978dfa2f8f6426e807e96e45a32168584927e33c0bffab8fe4b6de714,
092d550428f8497aa475ec5fdea54c42d83298ac1c9a6b1ee035094eadb9b260, 853a4ed8c6a9fa410b4d115916941f591980a6e0b8bbf06a5bcd9d87e3cc84dd,
11f304986e967fca41ddca87018ee4322aed89c899a59fa990f0b68dbb22671d, 4e118ae8f6f564a2991f5b1bdd7e57f6d5d4513d87921322838c7c2d9609c325,
c46488181ab9a92fc4c584984ee4b304a00a1fb827ac2e9ae0228232e8c33912, d9a0cb0db881836b9096c68dfc44a96e342054d79073070cf1d9dddfbcc01b3b,

6494f7f84f9a0f86c7ed65cbcf270c05fd431c8ef3ceadcb8500aaa537eee53a, 4549d02bf830d3c0e3abbbbc6dc4e4b40884ee30aa83fa69b25ed8edb85f7fd7,
2d39cab5f2aa58c18616cdba8e333b1c582415cd54849ca3d00933a69d164025, d9a0cb0db881836b9096c68dfc44a96e342054d79073070cf1d9dddfbcc01b3b,
d234993129d0b2fd918bcde0f3b582e197aae3e4ede833a519a811a696cba098, 9512dab11ee3518f8dd914dc49acce549a751d924f707125cc33b304f2548f5d,
0901e6e32f647001faa1938be2d8c34cb38c92da0d05f1c0e25494e4e03d2770, 418f2251bcef262087c1b6f05a4d07d6afa4cb3f7ea11bfc5e297e590013730c,
f3f802f51f47b4f882aac7460e780ace2b32ee70f409f87a47301bbd3bbc3e95, 17c0ae61e915aea6dbd083ce5c40b0e6dec8f228407ea44473bf659321ccb8f0,
2ef132abf949b8352da49915915da8607a854bb1258c2255d0ca62f8d55e5f76, e1396dc97410496fe2d70710b83fcd00ac643e8660003a7e5053afb93085dc54

A.2 Apps Used in the T-Recs Evaluation

This section presents hashes of the two sets of apps used in evaluating T-Recs explained in Section 4.4.

A.2.1 Popular Apps Collected from Google Play Store in 2016

Table A.4 shows hashes of 96 apps used in Section 4.4.3. Note that Table A.4 contains all apps shown in Table A.1.

TABLE A.4: Hashes of 96 apps collected from Google Play Store in 2016.

c2319fd61edddb8f02c7f0656424a72bce98bebae60585488b04b21281c34578, 179af04e39e220e908734c614d981492e9847a9260e3033cc8e97b61ac176ee9,
57c17f2076584b8387dea11fee6a08fc015c3acd1f666dbed573779508905c35, 1f54ca7489f7263294811cd2689a14506fc4b8cb3791fbcafa89ef08a8c1c5cd,
385bd2a3e0e6c99e175fe131bcad9db89e60c341538556aa784c7a8fe5990422, f8c6dc0d8acbacf7d091fa6d066d59d698a27868e76b9dc88f8ec621270b908b,
cdab4415a273d8e2f1779f8c22498e9a74929919f0a8388591f63a88055f8204, 0ad90fc12c3a647f01ec7cc6d41a295e673b3106903be7bc30adc98d33835b1d,
41126e3999cb4d7a7910716260a8255d7754e7ddc945c68de6a089b1c0dd8ccc, c94539ba3e4777a1346bf595d74365fd6d5de6d6457d9b079f8b2d3fda3cab9a,
c7fbfb20eef40991713844ffc46ce074b527b354dc23225ad6136ab22b119c2f, 328663f3ee60c1ab9e0c915892634b9bc80c2bb1142f47d74d39a0c4eba118f1,
428171da64f395b93840caf29f3ff82078b1f091ee7aae762ee64fccb73227e8, 305f4e45d5fc1d5abae7563682f2bc1cd79d46b4f5fa06d6f20d4d96bf8d4d25,
9f72c7231285695695019c5407e65d0cdb92d96cbdd32c37b506e6fb17ff91a3, 24e22029cdab3d406fc4d743e362d11b4fd897598a0bac8abb583edf810e9ae8,
35f51a54e3c8c27ee6804117c234cedfa37de7a3ddd0ab53049c308002f1939a, 1d6bfdb5bfe601b5a35cdd0c7be2320d5cd46180584fd51243b292626990dec0,
9c4e0807ac1d830786dcb9b6731fb08eb047d81f1a42711471182205b8ea72df, 4980855fa6771ea680e070e944e802d34f7ed8f8e1e990ce781980a9e7591d1e,
66a1aef7e5b94ab8adeb185baaf2f27097c6c75059982ba81c6472ac9c7c449d, d84df839ebfaf0c94979b4baae7b31d3dbc37bf5efa3a7ef8c80c65955447a7f,
e9ee29e9a92316e515ef226f9dfc75bab96dc86c183b680bf45b2edaf95ce5d5, 81f10d4313335521a2aa5c6eb18e9ed7030a054c69aef36b9b56a105b7fb0d99,
5d2a0f4ce04889e7f9c4ece7a525551577de064aed9e2f1cf5166db4479b84f1, f7e336926c246dc2f3dbb831671c6154ed5f4bfdd99374a2209fb5fd83ddd8b4,
e180003d940b8675c1035f0e8278399ee544936921b29b6b77f212dc46fd1ac3, 55807711bfc5ea942308b446ad6a6952d4756e7bcb0511dc819a8ad50e128e35,
f33e3fd3e3a020addb9c7edfdf0b3b13bc382221dc1c967a81a7de702dc239ad, 21f8ade6e7d910e56dcf2b32e020bba228befebd2d7a07e44bc0012d5bbb1323,
e808b39c89a89dd4b90f1d610ad761f37afeb7b91117f0d57a07c84acc4e5caf, 5cc2c54283a13d1130ce341ec4e91a5919f83635cd2e238d88517238272eaf34,
2183e6282243269d7ab25a31b5f0b1fcfd15d73b0e24890fb5bace8c19657f53, f11d0ecc84f4731b2e93ee049997d925cb802a5b86c1d30efd0ae6bb5132019f,
5dab984ca5fab1f1cf54eac10a2d3bb2802c9dece336975792a8081c083d1a12, af7a3cc40ffb6930398048a86f1a2c4cd1368c966e5826534a7118a89d45eb28,
4cdbc97b6a0407b7f1fa2164815d10f35af6e91d4a8e5d93784f7aecf0f97175, 73c4ffaa5ca0e954e1443b6085e536a83219e9a2e5d15292aaf1188a38592cfc,
b5d091f701ad668a49a37be6c6e754cf2424f037bbe838f237178a2eb09c545b, 42adb7a21602e20d4fa060b58071bd148fe1f15506a39dd85935b12064497a3e,
0104a9cd91c8ca399b1728af04c3d9a5f543017028ce12b2c935af89bda8bedb, e9c4e909fdfd09dc5f38897e5ff968efa571ff6d2f31e105b3eb915abb690fe8,
2ab6ea68266066f476bc251ebfc5ac7e510bde98d3706d74d3feb5bc9dfce0eb, 4e7abb31e72a286447ccec93ea5604bbbbc5e43f273d3301f312c564d4071c02,
235010d4454a3dce9b334f99a26ddc7d4123737363a43823c19d96fb970ba00f, 7600e01d616401987dbeb194adabe4bed0906fcd0f2063eb34f5a7f3cc6441dc,
8ef787a274f76b912ae2db389212ecf0cc0cf9d13c9cc34ec81f2b322a0404f5, efb39a8b64400c285ff41c90e0abb734c2f76059c8ae9b6631e533c841c7d7f6,
847192093ef1fe98f0bce9f6ed24526ba49ae41392dba04c9e0395fbf92320ca, ddd767453987ebac2051f23b34000b68f60e6ec5cfc323a9dc9009d2874c789b,
73d6b0e87385e23f9ac6a58b19cc4cc840403c23ff7cfc7f4cae499859eb8a6f, 32b17a1d32d1153c14afabad0da5fb371735d1eb33e972da1ec6a8921f80f47f,
eae11c32fadd6fb3eb171d1cf78aec84ec537a35bc15648b1a7b91b4af09deb2, 8676f515d008e5b35be1273af5a6a5abd77cbead9976fa92b4e5c61d90d95c6f,
d382980561f78d4bd88b6da9618299ea975d7ec1fbfa4e33818a9dc751e588cd, 35c86dd4e5cd693c4efb18157e4357291aa0395da65428a678122d93ea422da0,
86501d9aa43b0843c19e29916f5c29ea9ba69724c9de48bc256ea0646686dcd4, 01c7213b17bf459f5b43c8e537e4f675da3292dc227d8ef241a95fb6ed5c98ad,
18b7b4917261614223e887b357fd4a8a150156cecce39f6a63582fbed83c76ca, 452a42dadb3865804f7c85cd24cd1af85c203980581e41c60e3b6b4458ae36c0,
15b51b8ace8fd51a92e571ea412b9774eb6f004c730ced69b9fbb1a732ba3810, a14c4158898656abc08bcbf43df9299c5dec984068460d5826f2e8ef5f0b7c75,
12b1832c0ebd9fc13104b114de22db730bc37cc0d87bc82dc8b3144c44990736, a16ed39c64a7cfb692a1b42757ce5bb85dcf60355ca4b98109e3445fdf5d23bd,
668047b8393391a3682e5ef429164124f2990f39f9dc3c9b130340862fd87436, ab4e31691b5625c99ed85df6f900b9a0c8b0b822dd1df7c2265662ff19c21bbc,
44ac462f846cffcbe3d3156d00c9837a30228b12fa4b2287bfdf1767edbda3e7, 6612bdb546869a15bb368cd959b0fa268dff7a296af381f8c95b63ab4e07de5b,
401722b99fe626a1ba6bbed8f5809135dbed706c061315d308e7284e4dbb91a7, 202c070d4e131e1585beafd7d9de38869b9b2d584656d2783ea2013d8a32f49e,
92ee7233b6b6febff143a758c812fb51040496c36ce7c30b7dacd96d3525a6e4, fe303252e8ee6429097b79ddba48b220ad20e42ac64bb4a1e15df1a69e8cf98f,
b68bacbfa82639db8c7a2a9b767dd88643ff044ca869ed6e58792ddd4ff3cbf1, 9f4e11f58bc11a7691852366eac105a6cebc286b1af166132065e8241271b114,
e521b71a39e450a0a9a386aee1e9b3fc899cb663ccff94ae68805094c1df8d3d, 9d919ba99629b8a061aab253135c186167b2ea6123d7d3830654869cafccd5f4,
8cf04193f124693a5643dcbaf1f0b969f9a481c6ff749acad8a1bd8e15fdb1ac, db853b9f2cafadfd5aedc9526de6fa597bf11994333e4e0eeb68c474adca4116,
77930c6c5ec16e72664a95ebc00d36f682e37bbb9d4caa02cda2495ab3fc1831, 554730324c46d55c0ed84c74449b1faa9694f59ecf9c9dcaa161a0e6cbcbd7ae,
74181fb4e61f07c679250a8d963cf535fda0d0e4fd2db642d460a49226f29a56, 0dccd2823165f563ce101f66a0b290b5d739a35035d8993fae4be253b7c53058,
dcc8c3616f020c35375d3c99a37fa1e5b504e8a2e6b4e7d6491d720db58d945b, a87798d319bdec04b6331a5feaafab2f23b84c6e90ea45bc1183118e7615ed2c,
fcef2e5635cc0baebbcb9d4cad61b2afec163a9f8da7dc23e728e2e2912990f9, 0002e85941e05cede3bafbcc9cd1f5d2c40b0b9f05e49e90074c24c5efd132e2,
9dddf7fe33db82640fac5e15d8d65b71ef1158066e326e54b9a4291add9a35b3, f5966cf6ff044d26fe49fd8906dab23e81cb9c9aac42c33bd49925549db1d105,
0537de0f53edb736898da4919edb8e01c3049be1ee4dddec8b8f005323e5e86e, 98f7cbd948ddf41b112579600535da71f748394c1a6f61e9ebfe6a158f5291ed,
7905242ba51fe30824b2cc3ccced60c63a1bf49a2065f7024a399c0e51572e68, a44626cbdc5d85d230d3afd7be29dd660d53e911e88fcd44e87e37145d462183,
d1eb0e7e4faf16832adcbf3f2ac3341e3a4e8c98b22ee24b9b52526f0c11b82a, 2e0142dfb03c2f0c42d103ef52089407e34ed13ab2cb2ade47de8b0788108514,
548ed9f19181381c0440a1ef07937211a253799a32d92f77300967525da63a7e, 703b0f193ebc9713da8d56dcc3bf668683690e92d0ed7ffa8ce9a85f9ba6d405

A.2.2 Popular Apps Collected from Google Play Store in 2021

Table A.5 shows hashes of 158 apps used in Section 4.4.5. Note that this app set is a part of the app set shown in Table A.2.

TABLE A.5: Hashes of 158 apps collected from Google Play Store in 2021.

93743bdd9ec9919446159fbdc4716505fb13776a5423572f85cd31887fea6057, aea9687feef84d6c5c45ba6828520b45d9320a50b8acbb3e00898b29b91ac751,
99ed3082155257df8def811c47b6fa1980a31750592dce33b7a22dc0fad50e45, 4db06f56dcfcec21cc1808e8954bf5272bbef4d2237a822963324d0d76b62a5c,
e53b89177e697eff71776ab360727afee73f58e4492cd758a8cb47d3618bdc12, 3ffe2a52eda8f4eb3cc90d599aa7252bfe0759d1e7e65f23221ef007797fa947,
882ca0bd7bc58d380ca3fcae3510891eeea4475e37728a065e43d5d9013a5fed, 7680b4db799fccbfd639efefb6856b4e83a20c58f0733a52abcd565bec76ee85,
4c5e1236595c81a04dbee97d374109e4f4df9cec990bc514bb2098e364f73e00, e39ad0baee1d61246e17f7d3d213cf2c65cf3ee41c6851230f13d50224bcf0e4,
bba3d239fddd7bb1bc7b18204cfc0380a1ce519ffd08716cd65184e8cf86bd66, 8c16ffae75fa7609626feb15b317486286bc752fb2dd9e05e3209758054133c4,
976f247bf4dd445d7b51f5db2daf439e1eb08297b6a4432332425f7cbd6ec27e, a595aff30bb60a8a8b583af6998fd4d13edc9ad4f6bff19b194d1c5d6d459e56,
dd7a2833705d79867531ec660e1c9f441156fc61381f292f13bcad6b9677f6e2, 89faf29f3d81ded7e81d0adde7cac501f0ad645d68a420866f35f31a7a50f292,
f0e1f423dfb50a5a0c7135fded2fdc777e1e2fe14c9eb5df099271e9b2e534bd, 0346f2e43613fe50be16c9fce9ad3ffa689a50773c353f19171a3e51704c4809,
d318a6994b74960bb85e3f454af59b2f03605db83f19732c6d13928527848c58, d5f4ad05fbfafe15bfd5b5d6c2a294e0a9c250211a026fa090977f535d30eac3,
2f5ff3459c73a2d168e8cf5f3339047b346dc82a044e262dd6820d8cbafc87b6, ce5365bc90d3b750a85ba5fb8eadff09c22a9b70d200ef77aa8c0970f6c31d16,
b66e0b731dcddd0f167bdfde4c9b1eae31a86f242cb9f0be37988a7478eee521, e67f019b3aba4e8388837da735866b8e241882a2128a2db6d78a707d54ab31d0,
fe762b1bd71b2a9ee80cd1a85e189c21078b069fa921dd455e0286cf875abc10, 1b20f8305956e0ee245b488b2cb451746efb44793a00e03bb346280196c4c974,
ea9adc48032442f3779c04c084b2860d44cb16b5a33397b6c78ff59cc75b9fe8, f9bfa1680d1b62178ab27bf1d6829cafd9e1af0097d7ee22b3677bf666e7bde6,
f998a13609e1a2ca0ab2eb76180dc3ed8970dee2f3f72bccf7a3e1fa9742c226, db2c734d9041878ac99f32023b755a0af6566348e7925c13267588edc734d967,
a3a434d95f0fec2e543328c85cd3fef412dfc465169b41979762ddf507f90680, 9c463392057c86f0ccce39e93ffeb8fb2f225ecec31cb84c2da8b60884245efc,
c808ef7e416c7db772840e66a8416e07e13a5b08d2e064c556f65079fac9e4a5, f28ea1b9f085339b4c664562377e7926d6a999ca81b768fa8286c53b5be90fde,
5ceab8764de790b3550c653d736d483b56afc459a7a5e49bffacd786ac7e00ca, e352f9d9d74bdd0c3578296ae4e80dd7b3722d24568a44ea5889e9d5e13e4477,
20c224b1b12713b743326d89c4294bb54c6e47605a14395bf548a7672e42a3c3, 66044239eb7317c34bf1ecd8855bc26497dfdb32de68cf14b62f6916d74d1835,
99dc4b979f8499412a0e0f06d963f077f852b08bf066fbe28636105595da9d4f, 46a0aacacd6c403bafc692e55e71cf58ebf6a9b156dc5cf159b5a29c97f76d29,
b39f84e7146afed6074f96b90f56a2ef621b26e835c9b2c1dc72f7b43be594a5, 91eec64749ac26af48a94cc352534dc658b5394d395ff26f68e1091bb0caa25d,
25678f286172670a72478b3418c6d99689923e47ecc5f5aa888782546e1a0233, 0b9f81c77fb6161e0c69ca08fae8e63d59bf07216d4f5cb559909a8717ef6ed1,
c5e41c76f0074ce833219d97bbd16fe2caaeedd1fed92a16bf4e53c42ea7ceee, ee2dcd56258aee6f7e17c76b682ed72e8388f020d6dbcd0b7289c95fb2a691c7,
c669950a39ce73eb5336356bb32376a2a150a25e1472e98880a80e1fe279e65d, fe80a18fa969879b5d8b0369d4d84f21ffaff024d4a20c572ab686b099df4b18,
b7149ce223d8134a556099812f5af8a3b75f55ca511b9c2e3a4674dc1ec36d8e, 4a1c5ae33cc3aeed744a51038df821f617c2da419f9f50fe9f709d28821ead03,
88c8201b80ae5cf7cffd71f99d723a0a67668bef2f6b1bc03246cadc2f749c65, e9c2bd1bc41d47b166f4020ccdda248263052c2ae4fdf09862ee7d7e8082ae31,
232022fa86dd20f310f096122e48472d9e8eeaeb1629a570133f436ca76d5edc, 66e2c24f55892790fdd042caf1f47bf0b16d85bae26693852e8730d6a8580ec3,
9a1fe7ba72f9a04c1c1e5e7310301cf7a111dc2da6902e5deb302d0e119c508d, 948bfa4ab9ca4794394ffb156e6802856a7ae36933f91dd29109d59ad8210699,
a3a9101b94fae560bb2b7d1ef3c8616bc7771a44ca9c8bfd018eda3fa550b0b3, 832bed4187cc470e03c2db4f7e42f5adb53d68c280879e701103b6bee3a0d8ba,
5709d451a6de4a070643721f4596b6bfd5380f26da7a27d1d915f93753aeaee4, 769c60e4f420f13aa2c3ee62337a73f74011d5eaf8892b3ce873561e9c5a24cd,
61de7e4f717e13d7fd2056c2e222a8e7b9e546babd528cfe6b2c1852972c15d6, 61f1c706b4aa72a9e64a25fd2a082181487969a373020c37a176800a1beb99fe,
499427f1bd3c085082ffc1c23faf95540c6006d7daa70463aec51bb042ac2e8b, b2605b21f6eff598c7b446a6cf4fe13ce74c6befdf2069aaaf859a967249cc38,
7859739908505860df5166acac3c2e9f5e44c1ea4258a0ccfc2fd5e9aa677e90, a5113a4b8746b7ffd7d0b6d2f9fa77714a367c891518bfd341499364e4b7c0cd,
8154022d610dacdea08c7419f7baa45b4b62b2fd0533ff0d5bb5d6822d028110, 5aa93a29dff4cd520e69c87e93ae32fb69aebe467227b5354442fc171cd92173,
463681c66ef6aca7ad1847b98fe3ec0ee3afbe649f8740b0a568180e7ad72d95, 4ba1d318683e2711d5d98aadb368676b93b9b161ede80b84f3b4e99cf823a706,
7a7971d366a136f81e7ab43787184ebafa3577a2c40c1df413c199f1d078253f, a0ec014e633b1a0def25d523ecf81e5530e091be1c9075fad7af8aee97c86038,
389c7dec20d3cad1a873ada95c9036ef8d6e882023d7a1d0bc3b962c76dcdd51, d827419548306569624l8f043bbef70aa7e920499b638e4c35b66100f2c32d95,
c1e64bd45bb1f812564c4fd788045a1f266d807ba95da7c9cae1c991e0d367b7, 8a78d9c87011e54dd2fdf06c535b664544826510c8d8518b088bb992a36966ed,
9799204773c27134d929bdb7c4f026451703717bb7c25a8668727a7212ecfd29, cf43f249f89390dabd5bf53b906f9d5fe20c5575b0313997bd7109d61d8418eb,
5648004fc4788b2dee4a64151f85fbd7e9831b82b9336e1bbbeacec0aa466827, badfb2b6d143c886f10b71219e384689aebf932316cb8af0480e5f9ac32a05b5,
4c227ce66be3f17ea2c76b646263d943c5d1859254d717e97494c8647e1b1de1, dac4ff226287d965ca127ff25ffc792f8c29baaa31392e29a4fbbc48f22e4fea,
fbc1432e9c0a84e9e878452825c3acaf3efbf9ff1fe489c4398e5d0ca7309370, 76e970b5d30f6dcb89225c31d4004b3fc683ca58c86ac148b139a3638323e089,
175f81d73067caf0837d8a38a3e394bd74993e34e2658537b072814cd87785ac, 1d6239f165f9c40a6dd36054fe99768272bc62a2054fa1c30f5af8085b36d902,
a296ebfe5c73251000bb0769569951336ae7d1cb6f443a2cc9d63f111b8c94d2, 1d80a31879e87798cd9d578a8f510494bde6b7c43cdbc5ea326ce59f719ee327,
c8b7046749d78a748dadc144355b1a21fe1c75317dac2818e9fe777a53f66ee5, e4e5174c1f429d198825d8bb72262a02c4050fca8df8ec59b73b2fb18b196e89,
2205316c04a1043e670fa9a8ba7f1ff427b8c9b7e0e265bec0bf32a3456521d2, 62100bd3a9ef310b12154c0c69be50d37ac3fa28871aeb45a289502525f2477e,
464b61c19f409bc7035294b50df365ddab4c61d5c701c175cf56d023ff5e927f, 389902d32f67555f8bb9c959c3654e691c680d24e20b5b22698d2d9407e3c561,
c9b33c978ed9e3f65e71c451356226ed5fcc09480ddfa3b3c2872d9ef4a13a66, 3e56aa0a0a38f49a7f2e84bd49ba3112d757a84b3a15a36507676acec64c78df,
3328b30707e39aaedb06fcd228cef56dc8875b05458f9e8b9eb03388bf1f0af8, 5303bd1ef5c54b43190ada36b47b6f14baf0d04a5ad93a184d3b8bedc2aa6ab7,
3093cafc048cb12699f9064ec50ca98c22e7be4aeee563b796a07ed8549fa561, 20439fa4a6dcbdf1ec975cae5417499a448e997d8aca32ba809b0c79735e88c5,
f2e0c441a8f664aa7acd7acce35e861398a3ad63c208a4cd0ae9de056ae9645e, f35906d81b7e54a0c8b3c411a8b09f3e56a9d111a3999c9233b5fe61626e81af,
7a103c4ee48a9c2262388ee818303559f62967efad5c09862ea22fb913cd9d9e, 859ae9040b9925c9479466075679fda54f7a7a48195e019a5c13a44b9e23ab10,
c01010528305b60a0a3ead5e74416fc540aba880f7c8f8e9029d5b8a1a57f8ca, 919977e9bbe0c2b41e5e91b8e4fbfec8d4f22552d1e653a8a5cf038c01886349,
9936955360f4a06d638d18be5a8303486b82d27dd102a53e842d42a353ee079c, 3ceee2eb7bd089d0204f9d89a43e4de15d5f917bd9c8be87f46d15266e352a3b,
4400e2541abba700fd151bdf6aba01f1b5d9f6f80a5038acd6b37fbc9d2cbf59, 694d6cb8b81f030aaefb7f9c933b9bed8db177efb07c30e3dc8d937acdb8ae5c,
1fde6f3ac7f84b86a5a83ff968840cd1ee337c637e816c094bc3f006f539bde5, 4300a74faf8a1ea70c860855ce0a04def4dbc354f4e5940869bc17e6726dc821,
91ccce773160477e02eed7562e7ed9ef97c24a5938e030b328f4fee83bd789e7, 1afdc983261fff19101c2dfa54b551e274cf007a8ceac606f92ec509db11ac3c,
3933c8ad78a684ae2596d95ccfe987ed7c57b4ff8da016c9a1cde13842229234, 91d785e0ee4782e1a0a2016d8e0df5ec698ad4fef21913405857877e9eac9513,
b0d3a791ccf357479924d8a415910bf0cdcc7cfe1e89c3efbd8a743858854b7c, 7589736fb1edac7ff6018d1021592b3f2abaa8fac0313a18d5315046a268bef1,
92e59fbe8337edc0e55b35bda0bd95dd26da82503b4ce3cee80be570319c7fe2, 3ce537999fa73774ee8bdb181bf7d8e35b070b8aaa84094f80f2d30b601b3641,
6e57851e00491bdee27fd986c14a6bbc83584c775bba51f9dd59997ef853ccdc, 4cb502f81fba5f0bad91d2431d9f38b9bdf5b3f43ba5150547e23143038f88cb,
75fa06aa13657418fb8aeff47db7650b2f96943955cb5cea25974c2ed4a7c673, 9f4c9d990a2d2a126775a150d59cd7d05efe54b8a81fe07ce6dc61c0bbe40fb4,
253a6a66dc5a53f183baadc2580fa8cdb923d785669153a4d1f2deac8dea0c5c, 7a34e40c2e0a1b1290095677cdd8878b89309ecc3652b7f4c436688312d4d21a,
e73e1e2d86286c4fb51c57e4ee0f2c4d091174c3658170844cffaecec093ace4, b7f6e9d744798b8040c5ad6631dcb5c25c0a28123364aec8d3b68aeeaf392286,
fdfd7208a48a559959e705806e55784958e571dc97d4570b9a58b51ac0ac9dc7, 0ca75ac7f632954b258a5e4ed504eb2882f2ba707a81b5cd8ab567fbbae12e7e,
a2234d128d1c260e34ba7098aa15c68126d8e077c51f55bd16724b5867b3fd34, df2eda61d01519febfc1d86f1aeec9e1a684ef6df5c612bd79ab6d15e1520f95,
2a0d0ef7ddccdea78fd5ac94d2acce696fb3e2e3c7d4d66af59655baf29a73d6, 3771c601e10aaf060cf3f5e44fdc45e844950ecae8673469158b9737503c6967,
df3592ced9cb1b36953cb7d806fa56d3f580ea2d754c654ed67bdcd943173d4a, 76fd2604cb693d7ac93aaa55dab446bbc2e443a99d7006de03a0b2b9181f46ac,
69319bc8a6f296465e4981288104ec64954ba6c5b4c1b01159b4981eed559cca, a3d0f659a86bde22e3fc4bfb1db60637e2de715b63407ea4b49a61a0ed8bf83d,
6b6e04370ace1fd74ee88404acf3069285aed0d3d6d1a478f35af70c84e6eb1e, 9f57aedd77a352c89065dc5166e60670db7cf681d6fff7cbfa97d074d8d71762,

dc5767eb6ac4d8c31d2cb73db3b2237f08e776a35b84dfcedaef3310cc7bd659, 2af98ac89b277ba2a26e4211a751219407eb4a39a0fa07b8fa7d2f672eead7e9,
73fb069209933b5ac865869da56fd1ac55cb5145df3cf5bf27957cf7a9431037, 5e297f087b890f132ab5c1e59c7c278cbe18e54044a5ed57910d2812f55019eb,
3cc8e22493763c3f2c53609818fae023b6070f0cb7128ad429e387f836cddba0, 0ea9a33bdea21236ce91ca0e223f51b4abcba9489ea35c4c8b68775a18959b9f,
5a11c684f9cb6e59aeaa25400fe5e334c6a21763813e2e016d7eae458169cfd3, 0861826e6fefb273d6423b5000d89843ee8f9ddaa173d0c3010fd2e864c767ef,
d17413aaf5f15ddc0cc42a43b08735cc9329f89d34925f3b7ca54737b2fbde18, afd77edf2db2f99dd2220b1cc2b64ddfda40f3c23c39112194e7b60a28736a3f,
577acde222af8754bfc12c3524fe080225ea1dbc2d3a5b5dba3e1659634ab74c, 19dd32277da69516c2f737372c552d101b4779742eb0add8e41df9d20bfdb815,
d1d966589bb0f3b51e6fe9e5b3992ba4d898d65b3390e3d07ebe54bc17f1f66b, 541cb9b45045168255c2ed611cfdc84765e64408d94238b29dc4dc2f64f51399,
e8f16593db1aefaecb1442f0fa410314076a8dc7c96f79c54111da1e476cc543, fc544835cdf7b1a11bc09a039ae7a43ea827c6d9dd7803f8112133f598941291,
ba4808d35396a7b11a3f49cac3b962f19a49ea151c44812fd1fd87b919a4c292, 0b940f64f775f70fdbcf68c16d1d611326948810fb7d308dc1eb0e08ac5c43cf

Appendix B

Details of DroidBench 3.0 Analysis Results

This chapter shows the analysis results for each case in the 13 categories of DroidBench 3.0: Aliasing (Table B.1), Android Specific (Table B.2), Arrays and Lists (Table B.3), Callbacks (Table B.4), Emulator Detection (Table B.5), Field and Object Sensitivity (Table B.6), General Java (Table B.7), Inter Component Communication (Table B.8), Lifecycle (Table B.9), Reflection (Table B.10), Reflection_ICC (Table B.11), Threading (Table B.12), and Unreachable Code (Table B.13).

TABLE B.1: Analysis result for each case in the category Aliasing.

Test (#)	**E**	**T-Recs**			**FlowDroid$_{IC3}$**			**FlowDroid**			**Amandroid**			**DroidSafe**			**DroidRA$_F$**			**DroidRA$_A$**			**DroidRA$_D$**			**IccTA**			**TaintDroid**			**IntelliDroid**		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
FlowSensitivity1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Merge1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
SimpleAliasing1	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0
StrongUpdate1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sum (4)	1	1	0	0	1	1	0	1	1	0	1	1	0	0	0	1	1	1	0	1	1	0	0	0	1	1	1	0	1	0	0	1	0	0

TABLE B.2: Analysis result for each case in the category Android Specific.

Test (#)	**E**	**T-Recs**			**FlowDroid$_{IC3}$**			**FlowDroid**			**Amandroid**			**DroidSafe**			**DroidRA$_F$**			**DroidRA$_A$**			**DroidRA$_D$**			**IccTA**			**TaintDroid**			**IntelliDroid**		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
ApplicationModeling1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
DirectLeak1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
InactiveActivity	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Library2	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
LogNoLeak	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Obfuscation1	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0
Parcel1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
PrivateDataLeak3	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1	0	0
PublicAPIField1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
PublicAPIField2	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1
View1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Sum (11)	8	8	0	0	7	1	1	7	1	1	4	0	4	7	1	1	7	1	1	4	0	4	7	1	1	6	0	2	5	0	3	5	0	3

TABLE B.3: Analysis result for each case in the category Arrays and Lists.

Test (#)	**E**	**T-Recs**			**FlowDroid$_{IC3}$**			**FlowDroid**			**Amandroid**			**DroidSafe**			**DroidRA$_F$**			**DroidRA$_A$**			**DroidRA$_D$**			**IccTA**			**TaintDroid**			**IntelliDroid**		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
ArrayAccess1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
ArrayAccess2	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
ArrayAccess3	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
ArrayAccess4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ArrayAccess5	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
ArrayCopy1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
ArrayToString1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1
HashMapAccess1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
ListAccess1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
MultidimensionalArray1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
Sum (10)	4	4	0	0	4	3	0	4	5	0	1	4	3	4	4	0	4	5	0	1	4	3	4	4	0	3	5	1	1	0	3	1	0	3

TABLE B.4: Analysis result for each case in the category Callbacks.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
AnonymousClass1	2	2	0	0	1	0	1	1	0	1	0	0	2	2	0	0	1	0	1	0	0	2	2	0	0	1	0	1	0	0	2	0	0	2
Button1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
Button2	3	3	0	0	3	1	0	3	1	0	1	0	2	3	1	0	3	1	0	1	0	2	3	1	0	3	1	0	0	0	3	0	0	3
Button3	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
Button4	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
Button5	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
LocationLeak1	2	2	0	0	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	0	0	2	0	0	2
LocationLeak2	2	2	0	0	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	0	0	2	0	0	2
LocationLeak3	2	2	0	0	1	0	1	1	0	1	0	0	2	2	0	0	1	0	1	0	0	2	2	0	0	1	0	1	0	0	2	0	0	2
MethodOverride1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
MultiHandlers1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ordering1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
RegisterGlobal1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
RegisterGlobal2	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
Unregister1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
Sum (15)	18	18	0	0	15	2	3	15	2	3	2	1	16	18	4	0	15	2	3	2	1	16	18	4	0	16	2	2	1	0	17	1	0	17

TABLE B.5: Analysis result for each case in the category Emulator Detection.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Battery1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Bluetooth1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
Build1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
Contacts1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
ContentProvider1	2	2	0	0	2	0	0	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	2	0	0
DeviceId1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
File1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
IMEI1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
IP1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
PI1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
PlayStore1	2	2	0	0	2	0	0	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	0	0	2	2	0	0	0	0	2	0	0	2
PlayStore2	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	1
Sensors1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
SubscriberId1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
VoiceMail1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
Sum (15)	16	16	0	0	16	0	0	16	0	0	15	0	1	11	0	5	16	0	0	15	0	1	11	0	5	16	0	0	9	0	7	9	0	7

TABLE B.6: Analysis result for each case in the category Field and Object Sensitivity.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
FieldSensitivity1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FieldSensitivity2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FieldSensitivity3	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
FieldSensitivity4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
InheritedObjects1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
ObjectSensitivity1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ObjectSensitivity2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
Sum (7)	2	2	0	0	2	0	0	2	0	0	2	0	0	2	2	0	2	0	0	2	0	0	2	2	0	2	0	0	2	0	0	2	0	0

TABLE B.7: Analysis result for each case in the category General Java.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Clone1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
Exceptions1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
Exceptions2	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
Exceptions3	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
Exceptions4	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Exceptions5	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Exceptions6	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
Exceptions7	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
FactoryMethods1	2	2	0	0	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	0	0	2	2	0	0	2	0	0	0	0	2	2	0	0
Loop1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Loop2	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Serialization1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1
SourceCodeSpecific1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
StartProcessWithSecret1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
StaticInitialization1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1	0	0
StaticInitialization2	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
StaticInitialization3	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1
StringFormatter1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1
StringPatternMatching1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
StringToCharArray1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
StringToOutputStream1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
UnreachableCode	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
VirtualDispatch1	1	1	0	0	1	1	0	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	1	0	0	1
VirtualDispatch2	1	1	0	0	1	1	0	1	1	0	0	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1	0	0	0	1	0	0	1
VirtualDispatch3	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Sum (25)	22	22	0	0	18	4	4	18	4	4	5	2	17	22	2	0	18	4	4	5	2	17	22	2	0	18	5	4	10	0	12	12	0	10

TABLE B.8: Analysis result for each case in the category Inter Component Communication.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
ActivityCommunication1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0
ActivityCommunication2	2	2	0	0	1	0	1	1	0	1	2	1	0	2	0	0	1	0	1	2	1	0	2	0	0	2	0	0	2	0	0	2	0	0
ActivityCommunication3	2	2	0	0	2	0	0	1	0	1	2	1	0	1	0	1	1	0	1	2	1	0	1	0	1	1	0	1	2	0	0	2	0	0
ActivityCommunication4	2	2	0	0	2	0	0	1	0	1	2	1	0	2	0	0	1	0	1	2	1	0	2	0	0	2	0	0	2	0	0	2	0	0
ActivityCommunication5	2	2	0	0	2	0	0	1	0	1	2	1	0	1	0	1	1	0	1	2	1	0	1	0	1	1	0	1	2	0	0	2	0	0
ActivityCommunication6	2	2	0	0	0	0	2	1	0	1	2	1	0	1	0	1	1	0	1	2	1	0	1	0	1	1	0	1	2	0	0	2	0	0
ActivityCommunication7	2	2	0	0	2	0	0	1	0	1	2	1	0	1	0	1	1	0	1	2	1	0	1	0	1	1	0	1	2	0	0	2	0	0
ActivityCommunication8	2	2	0	0	1	0	1	1	0	1	2	1	0	2	0	0	1	0	1	2	1	0	2	0	0	2	0	0	2	0	0	2	0	0
BroadcastTaintAndLeak1	2	2	0	0	2	0	0	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1	0	0	2	0	0	2
ComponentNotInManifest1	1	1	0	0	1	0	0	1	0	0	1	1	0	0	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	0	0	1	0	0
EventOrdering1	1	1	0	0	0	0	1	0	0	1	1	1	0	1	0	0	0	0	1	1	1	0	1	0	0	1	0	0	0	0	1	1	0	0
IntentSink1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
IntentSink2	1	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	0	0	1	0	0	1
IntentSource1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
ServiceCommunication1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	0	0	1
SharedPreferences1	1	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
Singletons1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1
UnresolvableIntent1	3	3	0	0	1	0	2	1	0	2	3	0	0	3	0	0	1	0	2	3	0	0	3	0	0	2	0	1	2	0	1	2	0	1
Sum (18)	27	27	0	0	18	0	9	14	0	13	23	9	4	21	2	6	14	0	13	23	9	4	21	2	6	19	1	8	18	0	9	20	0	7

TABLE B.9: Analysis result for each case in the category Lifecycle.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
ActivityEventSequence1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0
ActivityEventSequence2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
ActivityEventSequence3	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
ActivityLifecycle1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
ActivityLifecycle2	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
ActivityLifecycle3	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
ActivityLifecycle4	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
ActivitySavedState1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
ApplicationLifecycle1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
ApplicationLifecycle2	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
ApplicationLifecycle3	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
AsynchronousEventOrdering1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
BroadcastReceiverLifecycle1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
BroadcastReceiverLifecycle2	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1	0	0
BroadcastReceiverLifecycle3	1	1	0	0	1	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	0	1	1	1	0	1	0	0	1	0	0	1	0	0
EventOrdering1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1
FragmentLifecycle1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	0
FragmentLifecycle2	1	1	0	0	0	0	1	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	0	0	1
ServiceEventSequence1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	1	0	0	1	0	0
ServiceEventSequence2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	2	0	0	0	0	0	1	0	0	2	0	0	0	0	0	0	0	0	0	0
ServiceEventSequence3	1	1	0	0	0	0	1	0	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1	0	1	0	0	0	0	1	0	0	1
ServiceLifecycle1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
ServiceLifecycle2	1	1	0	0	1	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	0	1	1	1	0	1	0	0	0	0	1	0	0	1
SharedPreferenceChanged1	1	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1
Sum (24)	21	21	0	0	14	1	7	14	1	7	6	2	15	21	9	0	13	1	8	6	2	15	21	9	0	16	1	5	9	0	12	9	0	12

TABLE B.10: Analysis result for each case in the category Reflection.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Reflection1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
Reflection2	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0
Reflection3	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0
Reflection4	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0
Reflection5	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0
Reflection6	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0
Reflection7	1	1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0
Reflection8	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0
Reflection9	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0
Sum (9)	9	9	0	0	8	0	1	8	0	1	1	0	8	4	0	5	8	0	1	6	0	3	6	0	3	1	0	8	9	0	0	9	0	0

TABLE B.11: Analysis result for each case in the category Reflection_ICC.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
ActivityCommunication2	2	2	0	0	0	0	2	0	0	2	2	0	0	0	0	2	0	0	2	2	0	0	0	0	2	0	0	2	2	0	0	2	0	0
AllReflection	2	2	0	0	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	2	0	0	2	0	0
OnlyIntent	2	2	0	0	1	0	1	1	0	1	1	0	1	0	0	2	1	0	1	1	0	1	0	0	2	2	0	0	2	0	0	2	0	0
OnlyIntentReceive	2	2	0	0	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	2	0	0	2	0	0
OnlySMS	2	2	0	0	1	0	1	1	0	1	1	0	1	0	0	2	1	0	1	1	0	1	0	0	2	0	0	2	2	0	0	2	0	0
OnlyTelephony	2	2	0	0	0	0	2	0	0	2	0	0	2	1	0	1	0	0	2	0	0	2	1	0	1	0	0	2	2	0	0	2	0	0
OnlyTelephony_Dynamic	3	3	0	0	0	0	3	0	0	3	0	0	3	2	0	1	0	0	3	0	0	3	2	0	1	0	0	3	3	0	0	3	0	0
OnlyTelephony_Reverse	2	2	0	0	0	0	2	0	0	2	0	0	2	1	0	1	0	0	2	0	0	2	1	0	1	0	0	2	2	0	0	2	0	0
OnlyTelephony_Substring	2	2	0	0	0	0	2	0	0	2	0	0	2	1	0	1	0	0	2	0	0	2	1	0	1	0	0	2	2	0	0	2	0	0
SharedPreferences1	2	2	0	0	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2
Sum (10)	21	21	0	0	2	0	19	2	0	19	4	0	17	5	0	16	2	0	19	4	0	17	5	0	16	2	0	19	19	0	2	19	0	2

TABLE B.12: Analysis result for each case in the category Threading.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
AsyncTask1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
Executor1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0
JavaThread1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	1	0	0	1	0	0
JavaThread2	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	0
Looper1	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0
TimerTask1	1	1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0
Sum (6)	6	6	0	0	5	0	1	5	0	1	1	0	5	4	1	2	5	0	1	1	0	5	4	1	2	3	0	3	6	0	0	6	0	0

TABLE B.13: Analysis result for each case in the category Unreachable Code.

Test (#)	E	T-Recs			FlowDroid$_{IC3}$			FlowDroid			Amandroid			DroidSafe			DroidRA$_F$			DroidRA$_A$			DroidRA$_D$			IccTA			TaintDroid			IntelliDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
SimpleUnreachable1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
UnreachableBoth	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
UnreachableSink1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
UnreachableSource1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
Sum (4)	0	0	0	0	0	3	0	0	3	0	0	4	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	0	0	0	0	0

Bibliography

[1] Statcounter. *Mobile Operating System Market Share Worldwide*. https://gs.statcounter.com/os-market-share/mobile/worldwide. (Visited on 08/2023).

[2] Google. *Google Play*. https://play.google.com/store. (Visited on 01/2021).

[3] Aptoide. *Aptoide*. https://en.aptoide.com/. (Visited on 09/2023).

[4] F-Droid. *F-Droid*. https://f-droid.org/en/. (Visited on 09/2023).

[5] Statista. *Number of available applications in the Google Play Store from December 2009 to June 2023*. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. (Visited on 08/2023).

[6] CNIL. *Mobile games: the CNIL fined VOODOO 3 million euros*. https://www.cnil.fr/en/mobile-games-cnil-fined-voodoo-3-million-euros. (Visited on 07/2023).

[7] FTC. *Ovulation Tracking App Premom Will be Barred from Sharing Health Data for Advertising Under Proposed FTC Order*. https://www.ftc.gov/news-events/news/press-releases/2023/05/ovulation-tracking-app-premom-will-be-barred-sharing-health-data-advertising-under-proposed-ftc. (Visited on 07/2023).

[8] Google. *Changes to Device Identifiers in Android O*. https://android-developers.googleblog.com/2017/04/changes-to-device-identifiers-in.html. (Visited on 09/2023).

[9] Google. *Privacy changes in Android 10*. https://developer.android.com/about/versions/10/privacy/changes. (Visited on 08/2022).

[10] I. Reyes, P. Wijesekera, J. Reardon, A. Elazari, A. Razaghpanah, N. Vallina-Rodriguez, and S. Egelman. "“Won't Somebody Think of the Children?” Examining COPPA Compliance at Scale". In: *Proc. Privacy Enhancing Technologies Symp.* http://dx.doi.org/10.1515/popets-2018-0021, July 2018, pp. 63–83.

[11] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu. "Toward a Framework for Detecting Privacy Policy Violations in Android Application Code". In: *Proceedings of the International Conference on Software Engineering*. https://doi.org/10.1145/2884781.2884855. 2016.

[12] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg. "Automated Analysis of Privacy Requirements for Mobile Apps". In: *AAAI Fall Symposium Series: Privacy and Language Technologies Technical Report FS-16-04*. 2016.

[13] D. Bui, Y. Yao, K. G. Shin, J.-M. Choi, and J. Shin. "Consistency Analysis of Data-Usage Purposes in Mobile Apps". In: *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* https://doi.org/10.1145/3460120.3484536, Nov. 2021.

[14] K. Zhao, X. Zhan, L. Yu, S. Zhou, H. Zhou, X. Luo, H. Wang, and Y. Liu. "Demystifying Privacy Policy of Third-Party Libraries in Mobile Apps". In: *Int. Conf. Softw. Eng.* https://doi.org/10.1109/ICSE48619.2023.00137, May 2023.

[15] B. Andow, S. Y. Mahmud, J. Whitaker, W. Enck, B. Reaves, and S. Egelman. "Actions Speak Louder than Words: Entity-Sensitive Privacy Policy and Data Flow Analysis with PoliCheck". In: *USENIX Secur. Symp.* Aug. 2020, pp. 985–1002.

[16] X. Zhang, X. Wang, R. Slavin, T. Breaux, and J. Niu. "How Does Misconfiguration of Analytic Services Compromise Mobile Privacy?" In: *Int. Conf. Softw. Eng.* https://doi.org/10.1145/3377811.3380401, 2020, pp. 1572–1583.

[17] R. Khandelwal, A. Nayak, P. Chung, and K. Fawaz. *Unpacking Privacy Labels: A Measurement and Developer Perspective on Google's Data Safety Section*. arXiv.2306.08111. 2023.

[18] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. "Bug Fixes, Improvements, ... and Privacy Leaks A Longitudinal Study of PII Leaks Across Android App Versions". In: *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. http://dx.doi.org/10.14722/ndss.2018.23143. San Diego, CA, USA, Feb. 2018.

[19] A. Bosu, F. Liu, D. (Yao, and G. Wang. "Collusive Data Leak and More: Large-Scale Threat Analysis of Inter-App Communications". In: *Proc. 2017 ACM Asia Conf. Comput. Commun. Secur. (ASIA CCS)*. https://doi.org/10.1145/3052973.3053004, Abu Dhabi, UAE, Apr. 2017, pp. 71–85.

[20] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. "Unsafe Exposure Analysis of Mobile In-App Advertisements". In: *Proc. ACM Conference Security Privacy Wireless Mobile Networks*. https://doi.org/10.1145/2185448.2185464, 2012, pp. 101–112.

[21] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna. "Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis". In: *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. http://dx.doi.org/10.14722/ndss.2017.23465. San Diego, CA, USA, Mar. 2017.

[22] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman. "50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System". In: *USENIX Security Symposium*. 2019, pp. 603–620.

[23] M. H. Meng, Q. Zhang, G. Xia, Y. Zheng, Y. Zhang, G. Bai, Z. Liu, S. G. Teo, and J. S. Dong. "Post-GDPR Threat Hunting on Android Phones: Dissecting OS-level Safeguards of User-unresettable Identifiers". In: *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. https://dx.doi.org/10.14722/ndss.2023.23176, San Diego, CA, USA, Mar. 2023.

[24] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin. "Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps". In: *Proc. 2020 IEEE Symp. Secur. Privacy (SP)*. https://doi.org/10.1109/SP40000.2020.00072. San Francisco, CA, USA, July 2020, pp. 1106–1120.

[25] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. "TriggerScope: Towards Detecting Logic Bombs in Android Applications". In: *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2016.30. May 2016.

[26] Y. Feng, S. Anand, I. Dillig, and A. Aiken. "Apposcopy: Semantics-Based Detection of Android Malware through Static Analysis". In: *Proc. ACM SIGSOFT International Symp. Foundations Software Engineering*. https://doi.org/10.1145/2635868.2635869, Nov. 2014, pp. 576–587.

[27] D. Gallingani, R. Gjomemo, V. Venkatakrishnan, and S. Zanero. "Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications". In: *IEEE Symp. Secur. and Privacy Workshops Mobile Secur. Technologies (MoST)*. May 2015.

[28] X. Cui, J. Wang, L. C. K. Hui, Z. Xie, T. Zeng, and S. M. Yiu. "WeChecker: Efficient and Precise Detection of Privilege Escalation Vulnerabilities in Android Apps". In: *Proc. ACM Conference Secur. Privacy Wireless Mobile Networks*. https://doi.org/10.1145/2766498.2766509, 2015.

[29] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". In: *Proc. 35th ACM SIGPLAN Conf. Program. Language Des. Implement. (PLDI)*. https://doi.org/10.1145/2594291.2594299. Edinburgh, U. K., June 2014, pp. 259–269.

[30] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. "Information-Flow Analysis of Android Applications in DroidSafe". In: *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. San Diego, CA, USA, Feb. 2015. URL: http://dx.doi.org/10.14722/ndss.2015.23089.

[31] 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: *ACM Trans. Privacy Secur.* 21.3 (Apr. 2018). Art. no. 14. URL: https://doi.org/10.1145/3183575.

[32] 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: *Proc. 2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng. (ICSE)*. https://doi.org/10.1109/ICSE.2015.48, Florence, Italy, May 2015, pp. 280–291.

[33] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis". In: *Proc. 2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng. (ICSE)*. https://doi.org/10.1109/ICSE.2015.30, Florence, Italy, May 2015, pp. 77–88.

[34] J. Samhi, A. Bartel, T. F. Bissyandé, and J. Klein. "RAICC: Revealing Atypical Inter-Component Communication in Android Apps". In: *Proc. 2021 IEEE/ACM 43rd IEEE Int. Conf. Softw. Eng. (ICSE)*. https://doi.org/10.1109/ICSE43902.2021.00126, Madrid, ES, May 2021, pp. 1398–1409.

[35] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. "Android Taint Flow Analysis for App Sets". In: *Proc. 3rd ACM SIGPLAN Int. Workshop State Art Java Program Anal. (SOAP)*. Art. no. 5. Edinburgh, U.K., June 2014. URL: https://doi.org/10.1145/2614628.2614633.

[36] 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: *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. San Diego, CA, USA, Feb. 2015. URL: http://dx.doi.org/10.14722/ndss.2015.23140.

[37] S. Rasthofer, S. Arzt, and E. Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks". In: *Proceedings of Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2014.23039. 2014.

[38] S. Arzt and E. Bodden. "StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework". In: *Proc. 2016 IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*. https://doi.org/10.1145/2884781.2884816. Austin, TX, USA, May 2016, pp. 725–735.

[39] 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: *Proc. 2015 IEEE/ACM 30th Int. Conf. Aut. Softw. Eng. (ASE)*. https://doi.org/10.1109/ASE.2015.69, Lincoln, NE, USA, Nov. 2015, pp. 669–679.

[40] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. "DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps". In: *Proc. 2016 Int. Symp. Softw. Testing Anal. (ISSTA)*. https://doi.org/10.1145/2931037.2931044, Saarbrücken, Germany, July 2016, pp. 318–329.

[41] 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". In: *ACM Trans. Softw. Eng. Methodol.* 30.3 (Apr. 2021). Art. no. 32. URL: https://doi.org/10.1145/3440033.

[42] X. Xiao, N. Tillmann, M. Fahndrich, J. De Halleux, and M. Moskal. "User-Aware Privacy Control via Extended Static-Information-Flow Analysis". In: *Proc. IEEE/ACM International Conference Automated Software Engineering*. https://doi.org/10.1145/2351676.2351689, 2012, pp. 80–89.

[43] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. "AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection". In: *Proc. ACM SIGSAC Conference Computer Communications Security*. https://doi.org/10.1145/2508859.2516676, 2013, pp. 1043–1054.

[44] X. Chen and S. Zhu. "DroidJust: Automated Functionality-Aware Privacy Leakage Analysis for Android Applications". In: *Proc. ACM Conference Security Privacy Wireless Mobile Networks*. https://doi.org/10.1145/2766498.2766507, 2015.

[45] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. "Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Dataflow Analysis and Peer Voting". In: *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. San Diego, CA, USA, Feb. 2015. URL: http://dx.doi.org/10.14722/ndss.2015.23287.

[46] S. Rahaman, I. Neamtiu, and X. Yin. "Algebraic-Datatype Taint Tracking, with Applications to Understanding Android Identifier Leaks". In: *Proc. ACM Joint Meeting European Software Engineering Conference Symposium Foundations Software Engineering*. https://doi.org/10.1145/3468264.3468550, 2021, pp. 70–82.

[47] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges, E. Bodden, and A. Zeller. "Heaps'n Leaks: How Heap Snapshots Improve Android Taint Analysis". In: *Proc. ACM/IEEE International Conference Software Engineering*. `https://doi.org/10.1145/3377811.3380438`, 2020, pp. 1061–1072.

[48] L. Luo, E. Bodden, and J. Späth. "A Qualitative Analysis of Android Taint-Analysis Results". In: *IEEE/ACM International Conference Automated Software Engineering*. `https://doi.org/10.1109/ASE.2019.00020`, Nov. 2019, pp. 102–114.

[49] W. Huang, Y. Dong, A. Milanova, and J. Dolby. "Scalable and Precise Taint Analysis for Android". In: *Proc. 2015 Int. Symp. Softw. Testing Anal. (ISSTA)*. `https://doi.org/10.1145/2771783.2771803`, Baltimore, MD, USA, July 2015, pp. 106–117.

[50] 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: *Proc. 2018 ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. `https://doi.org/10.1145/3243734.3243835`, Toronto, Canada, Oct. 2018, pp. 1137–1150.

[51] S. B. Andarzian and B. T. Ladani. "Compositional Taint Analysis of Native Codes for Security Vetting of Android Applications". In: *Proc. 2020 10th Int. Conf. Comput. and Knowledge Eng. (ICCKE)*. `https://doi.org/10.1109/ICCKE50421.2020.9303643`, Mashhad, Iran, Oct. 2020, pp. 567–572.

[52] 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: *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*. `https://doi.org/10.1145/3510003.3512766`, Pittsburgh, PA, USA, May 2022, pp. 1232–1244.

[53] C. Sun, Y. Ma, D. Zeng, G. Tan, S. Ma, and Y. Wu. "μDep: Mutation-Based Dependency Generation for Precise Taint Analysis on Android Native Code". In: *IEEE Transactions on Dependable and Secure Computing* 20.2 (Mar.–Apr. 2023), pp. 1461–1475. DOI: `https://doi.org/10.1109/TDSC.2022.3155693`.

[54] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *Proc. 9th USENIX Symp. Operating Syst. Des. Implement. (OSDI)*. `https://doi.org/10.1145/2619091`. Vancouver, BC, Canada, Oct. 2010, pp. 393–407.

[55] O. Tripp and J. Rubin. "A Bayesian Approach to Privacy Enforcement in Smartphones". In: *USENIX Secur. Symp.* Aug. 2014, pp. 175–190.

[56] Z. Wei and D. Lie. "LazyTainter: Memory-Efficient Taint Tracking in Managed Runtimes". In: *Proc. ACM Workshop Secur. and Privacy Smartphones Mobile Devices*. `https://doi.org/10.1145/2666620.2666626`, 2014, pp. 27–38.

[57] V. Rastogi, Y. Chen, and W. Enck. "AppsPlayground: Automatic Security Analysis of Smartphone Applications". In: *Proc. ACM Conference Data Application Secur. Privacy*. `https://doi.org/10.1145/2435349.2435379`, 2013, pp. 209–220.

[58] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. "Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis". In: *Proc. ACM SIGSAC Conference Computer Communications Secur.* `https://doi.org/10.1145/2508859.2516689`, 2013, pp. 611–622.

[59] M. Y. Wong and D. Lie. "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware". In: *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*. San Diego, CA, USA, Feb. 2016. URL: http://dx.doi.org/10.14722/ndss.2016.23118.

[60] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques". In: *Proc. Netw. Distrib. Sys. Secur. Symp. (NDSS)*. San Diego, CA, USA, Feb. 2016. URL: http://dx.doi.org/10.14722/ndss.2016.23066.

[61] Google. *Android Runtime (ART) and Dalvik*. https://source.android.com/devices/tech/dalvik. (Visited on 01/2021).

[62] M. Sun, T. Wei, and J. C. Lui. "TaintART: A Practical Multi-Level Information-Flow Tracking System for Android RunTime". In: *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. https://doi.org/10.1145/2976749.2978343. Vienna, Austria, Oct. 2016, pp. 331–342.

[63] M. Backes, S. Bugiel, O. Schranz, P. V. Styp-Rekowsky, and S. Weisgerber. "ARTist: The Android Runtime Instrumentation and Security Toolkit". In: *Proc. 2017 IEEE Eur. Symp. Secur. Privacy (EuroS&P)*. https://doi.org/10.1109/EuroSP.2017.43. Paris, France, Apr. 2017, pp. 481–495.

[64] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang. "TaintMan: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices". In: *IEEE Trans. Dependable Secure Comput.* 17.1 (Jan.–Feb. 2020). https://doi.org/10.1109/TDSC.2017.2740169, pp. 209–222.

[65] J. Schütte, A. Küechler, and D. TItze. "Practical Application-Level Dynamic Taint Analysis of Android Apps". In: *Proc. 2017 IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun. (Trustcom/BigDataSE/ICESS)*. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.215. Sydney, NSW, Australia, Aug. 2017, pp. 17–24.

[66] AOSP. *Dalvik bytecode*. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. (Visited on 01/2021).

[67] L. K. Yan and H. Yin. "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis". In: *Proc. 21st USENIX Secur. Symp.* Bellevue, WA, USA, Aug. 2012, pp. 569–584.

[68] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan. "On Tracking Information Flows through JNI in Android Applications". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. https://doi.org/10.1109/DSN.2014.30, June 2014, pp. 180–191.

[69] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. S. Chan. "NDroid: Toward Tracking Information Flows Across Multiple Android Contexts". In: *IEEE Trans. Inform. Forensics Secur.* 14.3 (Mar. 2019). https://doi.org/10.1109/TIFS.2018.2866347, pp. 814–828.

[70] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. "Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART". In: *USENIX Security Symposium*. Aug. 2017, pp. 289–306.

[71] L. Cavallaro, P. Saxena, and R. Sekar. *Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense*. Tech. rep. Stony Brook University, 2007.

[72] G. Sarwar, 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: *Proceedings of the 10th International Conference on Security and Cryptography*. https://doi.org/10.5220/0004535104610468. 2013.

[73] G. S. (Babil), O. Mehani, R. Boreli, and M.-A. Kaafar. *AntiTaintDroid (a.k.a. ScrubDroid)*. https://github.com/gsbabil/AntiTaintDroid. (Visited on 12/2020).

[74] W. You, B. Liang, J. Li, W. Shi, and X. Zhang. "Android Implicit Information Flow Demystified". In: *Proceedings of the ACM Symposium on Information, Computer and Communications Security*. https://doi.org/10.1145/2714576.2714604. 2015.

[75] C. Collberg. *the tigress c obfuscator*. https://tigress.wtf/. (Visited on 11/2023).

[76] M. Graa, N. Cuppens-Boulahia, F. Cuppens, J.-L. Lanet, and R. Moussaileb. "Detection of Side Channel Attacks Based on Data Tainting in Android Systems". In: *ICT Systems Security and Privacy Protection*. https://doi.org/10.1007/978-3-319-58469-0_14. 2017.

[77] F. Pauck, E. Bodden, and H. Wehrheim. "Do Android Taint Analysis Tools Keep Their Promises?" In: *Proc. 2018 26th ACM Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*. https://doi.org/10.1145/3236024.3236029, Lake Buena Vista, FL, USA, Oct. 2018, pp. 331–341.

[78] J. Zhang, Y. Wang, L. Qiu, and J. Rubin. "Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe". In: *IEEE Trans. Softw. Eng.* 48.10 (Oct. 2022). https://doi.org/10.1109/TSE.2021.3109563, pp. 4014–4040.

[79] Secure Software Engineering. *DroidBench 3.0*. URL: https://github.com/secure-software-engineering/DroidBench/tree/develop (visited on 03/2022).

[80] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, B. Wright, K. Butler, W. Enck, and P. Traynor. "*droid: Assessment and Evaluation of Android Application Analysis Tools". In: *ACM Comput. Surveys* 49.3 (Oct. 2016). Art. no. 55. URL: https://doi.org/10.1145/2996358.

[81] H. Inayoshi, S. Kakei, E. Takimoto, K. Mouri, and S. Saito. "Value-utilized Taint Propagation: Toward Precise Detection of Apps ' Information Flows across Android API Calls". In: *Int. Journal Information Secur.* 21 (Aug. 2022). https://doi.org/10.1007/s10207-022-00603-9, pp. 1127–1149.

[82] H. Inayoshi, S. Kakei, and S. Saito. "Execution Recording and Reconstruction for Detecting Information Flows in Android Apps". In: *IEEE Access* 11 (Jan. 2023). https://doi.org/10.1109/ACCESS.2023.3240724, pp. 10730–10750.

[83] Anzhi. URL: http://www.anzhi.com (visited on 09/2021).

[84] D. E. Denning and P. J. Denning. "Certification of Programs for Secure Information Flow". In: *Communications of the ACM* 20 (1977). https://doi.org/10.1145/359636.359712.

[85] A. Sabelfeld and A. C. Myers. "Language-based Information-flow Security". In: *IEEE J.Sel. A. Commun.* (2006). https://doi.org/10.1109/JSAC.2002.806121, pp. 5–19.

[86] J.-C. Wang, H.-M. Lee, C.-W. Chen, and A. B. Jeng. "Estimating intent-based covert channel bandwidth by time series decomposition analysis in Android platform". In: *IEEE Conference on Application, Information and Network Security*. https://doi.org/10.1109/AINS.2017.8270420. 2017.

[87] J. Han, C. Huang, F. Shi, and J. Liu. “Covert timing channel detection method based on time interval and payload length analysis”. In: *Computers & Security* 97 (2020). https://doi.org/10.1016/j.cose.2020.101952.

[88] U. Kargén, N. Mauthe, and N. Shahmehri. “Characterizing the Use of Code Obfuscation in Malicious and Benign Android Apps”. In: *Proc. Int. Conf. Availability Reliability Secur.* https://doi.org/10.1145/3600160.3600194, 2023.

[89] D. King, B. Hicks, M. Hicks, and T. Jaeger. “Implicit Flows: Can’t Live with ‘Em, Can’t Live without ‘Em”. In: *Proc. 4th Int. Conf. Inform. Syst. Secur. (ICISS)*. https://doi.org/10.1007/978-3-540-89862-7_4, Hyderabad, Andhra Pradesh, India, Dec. 2008, pp. 56–70.

[90] S. T. A. Rumee, D. L. (Deceased), and Y. Lei. “MirrorDroid: A framework to detect sensitive information leakage in Android by duplicate program execution”. In: *Annual Conference on Information Sciences and Systems*. https://doi.org/10.1109/CISS.2017.7926086. 2017.

[91] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. “DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation”. In: *Proc. Network and Distributed System Secur. Symp.* 2011.

[92] E. J. Schwartz, T. Avgerinos, and D. Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2010.26. 2010.

[93] E. Stinson and J. C. Mitchell. “Characterizing Bots’ Remote Control Behavior”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. https://doi.org/10.1007/978-3-540-73614-1_6. 2007.

[94] D. A. Lelewer and D. S. Hirschberg. “Data Compression”. In: *ACM Computer Surveys* (1987). https://doi.org/10.1145/45072.45074.

[95] W. Gasior and L. Yang. “Exploring Covert Channel in Android Platform”. In: *International Conference on Cyber Security*. https://doi.org/10.1109/CyberSecurity.2012.29. 2012.

[96] L. Georgiadis, R. F. Werneck, R. E. Tarjan, S. Triantafyllis, and D. I. August. “Finding Dominators in Practice”. In: *European Symposium on Algorithms*. https://doi.org/10.1007/978-3-540-30140-0_60. 2004.

[97] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. “CLAPP: Characterizing Loops in Android Applications”. In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. https://doi.org/10.1145/2786805.2786873. 2015.

[98] T. Wei, J. Mao, W. Zou, and Y. Chen. “A New Algorithm for Identifying Loops in Decompilation”. In: *International Static Analysis Symposium*. https://doi.org/10.1007/978-3-540-74061-2_11. 2007.

[99] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld. “An Empirical Study of Information Flows in Real-World JavaScript”. In: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*. https://doi.org/10.1145/3338504.3357339. 2019.

[100] M. Graa, N. C. Boulahia, F. Cuppens, and A. Cavalliy. "Protection against Code Obfuscation Attacks Based on Control Dependencies in Android Systems". In: *IEEE Eighth International Conference on Software Security and Reliability-Companion*. https://doi.org/10.1109/SERE-C.2014.33. 2014.

[101] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. "Provably Correct Runtime Enforcement of Non-interference Properties". In: *Information and Communications Security*. https://doi.org/10.1007/11935308_24. 2006.

[102] J. Stephens, B. Yadegari, C. Collberg, S. Debray, and C. Scheidegger. "Probabilistic Obfuscation Through Covert Channels". In: *IEEE European Symposium on Security and Privacy*. https://doi.org/10.1109/EuroSP.2018.00025. 2018.

[103] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel. "Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments". In: *IEEE/ACM 39th International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2017.35. 2017.

[104] J. Schütte, R. Fedler, and D. Titze. "ConDroid: Targeted Dynamic Analysis of Android Applications". In: *IEEE 29th International Conference on Advanced Information Networking and Applications*. https://doi.org/10.1109/AINA.2015.238. 2015.

[105] S. Chandra, Z. Lin, A. Kundu, and L. Khan. "Towards a Systematic Study of the Covert Channel Attacks in Smartphones". In: *International Conference on Security and Privacy in Communication Networks*. https://doi.org/10.1007/978-3-319-23829-6_29. 2015.

[106] J.-F. Lalande and S. Wendzel. "Hiding Privacy Leaks in Android Applications Using Low-Attention Raising Covert Channels". In: *Int. Conf. Availability, Reliability and Secur.* https://doi.org/10.1109/ARES.2013.92. 2013.

[107] D. Schreckling, J. Köstler, and M. Schaff. "Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for Android". In: *Information Security Technical Report* 17.3 (2013). https://doi.org/10.1016/j.istr.2012.10.006.

[108] G. Barbon, A. Cortesi, P. Ferrara, M. Pistoia, and O. Tripp. "Privacy Analysis of Android Apps: Implicit Flows and Quantitative Analysis". In: *Computer Information Systems and Industrial Management*. https://doi.org/10.1007/978-3-319-24369-6_1. 2015.

[109] Google. *UI/Application Exerciser Monkey*. URL: https://developer.android.com/studio/test/monkey (visited on 03/2022).

[110] V. Balachandran, Sufatrio, D. J. Tan, and V. L. Thing. "Control flow obfuscation for Android applications". In: *Comput. Secur.* 61 (Aug. 2016). https://doi.org/10.1016/j.cose.2016.05.003, pp. 72–93.

[111] Google. *Enable multidex for apps with over 64K methods*. URL: https://developer.android.com/studio/build/multidex (visited on 04/2022).

[112] Google. *Dalvik bytecode*. URL: https://source.android.com/devices/tech/dalvik/dalvik-bytecode (visited on 03/2022).

[113] Apktool. URL: https://ibotpeaches.github.io/Apktool/ (visited on 03/2022).

[114] JesusFreke. *Smali*. https://github.com/JesusFreke/smali. (Visited on 01/2021).

[115] J. Zhang, Y. Wang, L. Qiu, and J. Rubin. *Supplementary Materials*. URL: https://resess.github.io/artifacts/StaticTaint/index (visited on 03/2022).

[116] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. "AndroZoo: Collecting Millions of Android Apps for the Research Community". In: *Proc. 2016 IEEE/ACM 13th Working Conf. Mining Softw. Repositories (MSR)*. Austin, TX, USA, May 2016, pp. 468–471.

[117] Google. *AndroidX*. URL: https://developer.android.com/jetpack/androidx (visited on 04/2022).

[118] Secure Software Engineering. *FlowDroid*. URL: https://github.com/secure-software-engineering/FlowDroid (visited on 02/2022).

[119] Argus Group. *Argus-SAF*. URL: https://github.com/arguslab/Argus-SAF (visited on 02/2022).

[120] MIT-PAC. *droidsafe-src*. URL: https://github.com/MIT-PAC/droidsafe-src (visited on 02/2022).

[121] SerVal Research Group. *DroidRA*. URL: https://github.com/serval-snt-uni-lu/DroidRA (visited on 11/2022).

[122] Li Li. *IccTA*. URL: https://github.com/lilicoding/soot-infoflow-android-iccta (visited on 12/2022).

[123] J. Samhi. *RAICC*. URL: https://github.com/JordanSamhi/RAICC (visited on 11/2022).

[124] TaintDroid. URL: https://github.com/TaintDroid (visited on 02/2022).

[125] M. Wong. *IntelliDroid*. URL: https://github.com/miwong/IntelliDroid (visited on 02/2022).

[126] H. Inayoshi, S. Kakei, E. Takimoto, K. Mouri, and S. Saito. "VTDroid: Value-based Tracking for Overcoming Anti-Taint-Analysis Techniques in Android Apps". In: *Proc. Int. Conf. Availability, Reliability Secur.* Art. no. 29, https://doi.org/10.1145/3465481.3465759. Vienna, Austria, Aug. 2021.

[127] MIT-PAC. *Droidsafe-src*. URL: https://mit-pac.github.io/droidsafe-src (visited on 02/2022).

[128] Google. *Best practices for unique identifiers*. URL: https://developer.android.com/training/articles/user-data-ids (visited on 03/2022).

[129] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame. "You Shall not Repackage! Demystifying Anti-Repackaging on Android". In: *Comput. Secur.* 103 (Apr. 2021). https://doi.org/10.1016/j.cose.2021.102181,

[130] A. Mordahl and S. Wei. "The Impact of Tool Configuration Spaces on the Evaluation of Configurable Taint Analysis for Android". In: *Proc. 2021 Int. Symp. Softw. Testing and Anal. (ISSTA)*. https://doi.org/10.1145/3460319.3464823, Virtual, Denmark, July 2021, pp. 466–477.

[131] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. *SCanDroid: Automated security certification of Android applications.* Tech. rep. University of Maryland, Nov. 2009. URL: http://hdl.handle.net/1903/11847.

Publications

1. H. Inayoshi, S. Kakei, E. Takimoto, K. Mouri, and S. Saito: "Prevention of Data Leakage due to Implicit Information Flows in Android Applications", In: *Proc. 14th Asia Joint Conf. Information Secur. (AsiaJCIS)*, pp. 103-110, Kobe, Japan, Aug. 2019.

2. Y. Hayashi, H. Inayoshi, S. Kakei, E. Takimoto, K. Mouri, and S. Saito: "Effects of Compiler Optimization on Taint Analysis and its Performance Enhancement", (in Japanese), In: *Comp. Secur. Symp.*, pp. 990-997, Nagasaki, Japan, Oct. 2019.

3. H. Inayoshi, S. Kakei, E. Takimoto, K. Mouri, and S. Saito: "VTDroid: Value-based Tracking for Overcoming Anti-Taint-Analysis Techniques in Android Apps", In: *Proc. 16th Int. Conf. Availability, Reliability Secur. (ARES)*, Art. no. 29, Vienna, Austria, Aug. 2021.

4. M. Ohnishi, H. Inayoshi, S. Kakei, and S. Saito: "Improving the Accuracy of Constraints on Exceptions in Static Taint Analysis of Android Applications", (in Japanese), In: *Comp. Secur. Symp.*, pp. 952-959, virtual, Oct. 2021.

5. M. Ohnishi, H. Inayoshi, S. Kakei, and S. Saito: "Computing Constraints of Leakage due to Exception Handling in Android Applications", (in Japanese), In: *IPSJ SIG Tech. Rep. SPT*, Art. no. 13, virtual, Jul. 2022.

6. H. Inayoshi, S. Kakei, E. Takimoto, K. Mouri, and S. Saito: "Value-utilized Taint Propagation: Toward Precise Detection of Apps' Information Flows across Android API Calls", In: *Int. Journal Information Secur.* 21, pp. 1127–1149, Aug. 2022.

7. H. Inayoshi, S. Kakei, and S. Saito: "Plug and Analyze: Usable Dynamic Taint Tracker for Android Apps", In: *Proc. 22nd IEEE Int. Working Conf. Source Code Anal. Manipulation (SCAM)*, pp. 24-34, Limassol, Cyprus, Oct. 2022.

8. H. Inayoshi, S. Kakei, and S. Saito: "Study on Collecting Hardware Identifiers in Using Earlier Android Versions", (in Japanese), In: *Comp. Secur. Symp.*, pp. 532-539, Kumamoto, Japan, Oct. 2022.

9. H. Inayoshi, S. Kakei, and S. Saito: "Execution Recording and Reconstruction for Detecting Information Flows in Android Apps", In: *IEEE Access* 11, pp. 10730–10750, Jan. 2023.

10. H. Inayoshi, S. Kakei, and S. Saito: "Semi-automatic Detection of Inconsistencies between Guidance and Actual Behavior of Third-party SDKs in Android Apps", (in Japanese), In: *Comp. Secur. Symp.*, pp. 849-856, Fukuoka, Japan, Nov. 2023.

11. H. Inayoshi, S. Kakei, and S. Saito: "Detection of Inconsistencies between Guidance Pages and Actual Data Collection of Third-party SDKs in Android Apps", In: *Proc. 11th Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*, Lisbon, Portugal, Apr. 2024.