REDUCING FALSE NEGATIVES IN TAINT ANALYSIS VIA HYBRID SOURCE INFERENCE

by

XUELING ZHANG, B.S.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Jianwei Niu, Ph.D., Co-Chair
Xiaoyin Wang, Ph.D., Co-Chair
Travis Breaux, Ph.D
Amanda Fernandez, Ph.D.
Ravi Sandhu, Ph.D.
Rocky Slavin, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
December 2021

ACKNOWLEDGEMENTS

I would like to thank Dr. Jianwei Niu and Dr. Xiaoyin Wang for their advice on this work, as well as their continuous support of my PhD studies and research, as well as their patience, motivation, and enthusiasm. Their insights and knowledge significantly enriched my work. My committee members, Dr. Travis Breaux, Dr. Amanda Fernandez, Dr. Ravi Sandhu, and Dr. Rocky Slavin have all offered constructive questions and comments that have greatly helped me improve this thesis work. I also thank my family for their continuous support during my PhD journey.

This research was supported in part by the National Science Foundation under grants NSF-1748109, NSF-1453139, NSF-1846467, NSF-1736209, NSF-2007718, and NSF-1948244.

December 2021

REDUCING FALSE NEGATIVES IN TAINT ANALYSIS VIA HYBRID SOURCE INFERENCE

Xueling Zhang, Ph.D.
The University of Texas at San Antonio, 2021

Supervising Professors: Jianwei Niu, Ph.D., Xiaoyin Wang, Ph.D.

Mobile applications (apps) are widely used and frequently process sensitive data, such as a user's current location, health information, or dating preferences. Because of their access to sensitive data, mobile applications have made privacy a well-known challenge in the ecosystem. Users are usually unaware of, and have little control over, what and how their data is collected, stored and transmitted.

To assess the data practices of mobile apps, research community has made significant efforts in developing data flow analysis tools that can be applied to app code. These tools are designed to detect tainted information flows from sources, which allow access to sensitive data (e.g., the user's location), to sink, which are potential channels through which sensitive data could be leaked to an adversary (e.g., a network connection).

The data flow analysis tools require a list of sources and sinks as input and are generally classified into two categories : 1) Dynamic taint analysis, which tracks taint flow during runtime. The effectiveness of dynamic analysis is limited by the execution coverage and run-time overhead. 2) Static taint analysis, which inspects code without running it, is theoretically conservative and intend to detect all possible taint flows. However, it often reports false negatives due to 1) static inaccessibility. Static code analysis does not have access to the code that is only visible or determinable during run time execution, such as reflection, dynamically loaded code, native code, code executed on a remote server and so on; 2) incomplete sources, no matter how good the tool is, it can only guarantee data security when its list of sources and sinks is complete. If a source is missing, a malicious app can retrieve this data without being detected by the analysis tool. Earlier research efforts in this area have primarily focus on tracing the sensitive data extracted from

Android device through Android platform APIs with little works regarding sensitive data extracted through methods defined by apps or third-party libraries, which are also sources.

This thesis aims to reduce the false negatives in static taint analysis by uncovering sensitive data access through methods defined by app and third-party libraries other than Android platform APIs. Specifically, we utilize hybrid (combined static and dynamic) code analysis and machine learning techniques to detect such sensitive data access. The exposed data access can be used as 1) intermediate sources of Android API source whose taint path was interrupted by static inaccessibility code. 2) a new type of source that may lead to leaks.

TABLE OF CONTENTS

Vita

LIST OF TABLES

LIST OF FIGURES

Chapter 1: INTRODUCTION

Over the last decade, most people have used smartphones on a daily basis to perform a variety of tasks. The popularity of smartphones can be attributed to their ability to access millions of mobile apps, which provide people with a variety of features for both personal and professional use. As of September 2021, there are over 2.7 million Android apps available on Google Play [72]. Mobile apps may collect, process, and share sensitive user data for a variety of functional or commercial purposes. For example, most apps that require registration require the user's email address; a shopping app may save the user's name, credit card information, and home address; and a navigation app requires the user's current location.

Access to user's sensitive information invariably leads to privacy invasions. App users are usually unaware of, and have little control over, how mobile apps access and transmit their personal data. According to the report from Norwegian Consumer Council [68], Tinder, Grindr, and other eight apps were under investigation for sharing sensitive personal data with advertisers without the user's knowledge. Previous studies [111,129,147] also shown that users have a poor understanding of how sensitive data is used, and existing interfaces fall short of providing users with the information they need to make informed decisions. Data collection that goes beyond what is required for the apps to function, often without users' knowledge, results in privacy leaks and policy violations.

A number of regulations and policies are in place to protect mobile users' privacy, such as EU General Data Protection Regulation (GDPR), California Consumer Privacy Act(CCPA), and Health Insurance Portability and Accountability Act (HIPAA). They exist to ensure that organizations do not use or disclose customers' personal information in an unauthorized manner. Organizations must follow these privacy standards to legitimately access or disclose personal information acquired. Violations of these privacy regulations may result in legal ramifications and face severe financial penalties. For example, Facebook has been fined $5 billion by The Federal Trade Commission, for deceiving users about their ability to control the privacy of their personal information [27]. WhatsApp has been fined $267 million for violating the European Union's data

privacy regulations. According to the decision, WhatsApp did not properly inform EU citizens about how it handles their personal data, including how it shares that information with its parent company [74].

To provide users with the information they need to make decisions about using an app, regulators have mandated that app developers post privacy policies that inform their users about which and how their information will be collected, stored, and shared, as well as the purpose of the data collection and sharing. Each mobile app has its own privacy policy that users must accept before signing up or registering to use the app. However, previous research [117, 155, 165] has shown that few users read privacy policies because the text is verbose and difficult to understand. Besides, these policies may be written by people other than the developers, such as lawyers, who may not understand well about the data collection and sharing at the code level, the privacy policies they write may be inconsistent with the app code, or do not reflect the app's actual data practices [183, 201] .

To learn app's actual data practices, a number of ongoing research efforts focusing on identify sensitive data access and transmission by applying data flow analysis on app's code, statically [78, 90, 94, 118, 138, 204] or dynamically [108, 187, 215]. These analysis tools primarily trace the information flow from *sources*, which allow access to sensitive data (e.g., the user's location), to *sink*, which are potential channels through which sensitive data could be leaked to an adversary (e.g., a network connection). Dynamic taint analyses propagate taints at run time through memory locations so they always find true taint flows. However, they may miss taint flows which are not triggered during testing and will cause run-time overhead if applied during production. Alternatively, static taint analyses, propagate taints based on an overestimation of all possible program paths leading to the detection of all possible taint flows with no false negatives but some false positives due to infeasible execution paths. Despite the theoretical soundness of static taint analyses, various practical complexities often lead to false negatives in real-world scenarios [133, 143, 157]. Such false negatives may result in undetected vulnerabilities, privacy leaks, malicious apps, etc. The cause of these false negatives can often be attributed to 1) static inaccessibility. Static code

analyses are unable to access code that is only visible or determinable during run time execution, such as dynamically loaded code, native code, reflection code, code executed on a remote server, and so on; 2) incomplete sources. no matter how good the tool is, it can only guarantee data security when its list of sources and sinks is complete. If a source is missing, a malicious app can retrieve this data without being detected by the analysis tool.

To mitigate the issue of false negative in static taint analysis, many previous research efforts have primarily focused on handling inaccessible code, such as reflection code and dynamic loaded code. Various techniques have been used to deal with reflection: dynamic analysis, which uses refection information obtained at run-time to assist static taint analysis by connecting the class that invoked the reflection method with the class constructed by the reflection method [88, 191, 224]; static analysis, which uses static code analysis to resolve reflection, leveraging string information [139, 144, 185], data types [221], class name and member signature [140]. Dynamically loaded / generated code is typically handled by incorporating dynamic analysis as a supplement, utilizing the collected information such as dynamically generated code [206], dynamic code update [224], and calling structure [107] to supplement static analysis.

These works reduce false negatives of static taint analysis by resolving specific types of dynamic language features codes to complete interrupted program paths. They focus on filling in the inaccessible code so that the taint path is not interrupted. Other than completing the inaccessible code, few works have been done from the perspective of tainted data. Our research aims to identify sensitive data access that originated from the tainted source whose path was interrupted, and then use the identified data access as intermediate sources to detect whether it flows to sink. Our work differs from previously mentioned works in that it is not specific to any types of inaccessible code and treats inaccessible code as a blackbox, whereas the previous works dealt with specific code features that cause code inaccessibility. When applied to complex code practice in real-world apps, the independence of inaccessible types is a significant advantage over existing works. Also, there is a handful of work aim to identify sensitive data extracted through Android platform APIs [161], sensitive user input [126, 151] and privacy leaks caused by those sensitive

data access [79, 126, 151, 183, 201]. Little research has been conducted on sensitive data access through methods defined by apps or third-party libraries other than Android APIs, which could be new sources for detecting new leaks or intermediate sources of the sources whose taint path was interrupted by inaccessible code in static taint analyses.

1.1 Problem Statement

Mobile apps' ability to access users' sensitive data results in privacy invasions. Mobile users are often unaware of and have limited control over how mobile apps access and transmit their personal data. A number of ongoing research efforts focus on tracing sensitive data extracted through the Android platform APIs, little work has been done to explore the sensitive data access through methods defined by apps and third-party libraries , and see to what extent these data access could cause privacy leaks.

The fundamental goal of this thesis is to uncover data access through methods defined by apps and third-party libraries, which enables the detection of privacy leaks caused by data access beyond Android APIs, thus reduce false negatives in static taint analysis. We are aiming to answer two key research questions. The first one is, how do we discover sensitive data access through methods defined by app and third-party libraries? The other key research question is whether the identified data access can serve as valid sources in static taints analysis to reveal previously unknown privacy leaks, which is what users and developers are most concerned about.

1.2 Thesis Statement

Utilizing hybrid analysis (combined static and dynamic analysis) and machine learning techniques, we can build a framework that accurately identifies sensitive data access through methods defined by apps and third-party libraries. The identified data access can help users and developers in understanding what information has been collected, as well as reduce false negatives in taint analysis for privacy leak detection.

To achieve our goal, we first conducted an empirically study over 1,000 mobile apps about data collection and sharing. Second, we developed a hybrid taint analysis approach to discover data access through methods defined by app and third-party libraries, which can be used as supplement sources in static taint analysis for privacy leak detection. We evaluated this approach on both benchmark and real-world apps, and compared it with the state-of-the-art static analysis tools. We proceed to show that hybrid taint analysis can effectively detect data access through methods defined by apps and third-party libraries, as well as previously unknown privacy leaks. Third, we developed a machine-learning-based approach to identify methods for accessing sensitive data defined by apps and third-party libraries, as well as privacy leaks caused by them. We evaluated our approach by applying it on previously unseen apps. The results show that machine learning techniques can be used to predict sensitive data access with high precision. The predicted data access also cause privacy leaks that are not detected by program analysis based approaches.

1.3 Thesis Approach

Based on the objectives and the techniques involved, this thesis involves three major works. In this section, we will give a brief introduction to each of the three works and outline the key techniques we used.

1.3.1 Data Collection and Sharing: A Study on Analytic Services

For a better understanding of users' behavior, app developers often use analytic service to gather user's data. Analytic services usually provide client libraries that app developers could utilize in their app, which will record an app user's interaction with the app and send the corresponding data to the server of the analytic service. Later, the analytic services can link the activity of a mobile app user over time into a behavior report. The behavior report includes detailed usage information about this user. The analytic services can then aggregate all the users' reports and provide analytic data to the app developers so that they can improve their product or make better business decisions based on the analytic report. A major privacy risk associated with third-party

analytic services is the data usage after the behavioral reports have been collected by the analytic service. Once the data have left the app and reached an analytic service, the developers and users lose control of the information. Even if the third-party service is trusted not to misuse the data, accumulated long-term storage of user behavioral data is susceptible to theft or leakage [8, 27, 38]. Third-party services may also share the data to their business partners or do not provide enough protection for them.

To understand whether app shared user's PII (Personally Identifiable Information) with analytic services, we developed a semi-automatic approach to observe the data sharing during run time. In this approach, we first investigated the documentation of the 18 most popular analytic services in the mobile analytic ecosystem as listed in AppBrain [10]. Through their specification, we collected the Application Program Interface (API) that specifically for sharing uses' data. With this list of APIs, we designed and conducted an experiment to dynamically and automatically evaluate the top 1,000 Google Play store apps. We detected invocations of those data sharing API at run time and recorded parameter values to study what the common practices were and whether they abode by the app's privacy policies, the analytic service guidelines, and best practices of using PII in analytic services. We also investigated the analytic reports generated by the analytic services to study whether the services applied any mechanisms to anonymize or aggregate the collected data.

This study provides a foundation for us to better understand mobile apps in terms of data collection and sharing with third party services.

1.3.2 Uncover Data Access and Reduce False Negatives of Static Taint Analysis through Hybrid Program Analysis

Mobile apps and their included third-party services extensively collect private information. One main concern of mobile app users is the leakage of their private information. To address this issue, researchers have proposed a variety of analysis tools for detecting privacy leaks. The majority of approaches apply data flow analysis on the app code, either through static or dynamic taint analysis. Those tools require a list of sources that allow access to sensitive data (e.g., the user's location) as

well as a list of sinks, which are potential channels through which sensitive data could be leaked to an adversary (e.g., a network connection). Researchers have constructed lists of sensitive sources and sinks within the Android platform API. However, Android app developers may also store sensitive information within their apps or access sensitive data through third-party libraries, which are also sources. No matter how good the tool is, it can only guarantee data security when its list of sources and sinks is complete. If a source is missing, a malicious app can retrieve this data without being detected by the analysis tool. Incomplete source list cause false negatives in static taint analysis. Earlier studies [133, 143] also show the existence of false negatives in static taint analyses caused by dynamic programming language features such as reflection calls in Java, dynamically loaded or generated code, external code execution through database servers and network servers, and multi-language code (e.g., native code and shell scripts). In static taint analysis, these code are invisible or indeterminable , so the tracing of tainted data will be interrupted by them, resulting in false negatives.

To reduce false negatives in static taint analysis, we proposed an approach to identify sensitive data access through methods defined by app and third-party libraries, which can be used as intermediate sources in a interrupted path, or new sources not included in the existing source list. We developed a novel approach, DySTA, which uses dynamic taint analysis to observe sensitive data access and use them as additional sources for static taint analysis. However, naively adding sources causes static analysis to lose context sensitivity and thus produce false positives. Thus, we developed a hybrid context matching algorithm and corresponding tool, ConDySTA, to preserve context sensitivity in DySTA. We applied ConDySTA on both benchmark and real-world apps from google play store to show that hybrid analysis can detect data access through methods defined by app and third-party libraries, as well as privacy leaks that missed by the state-of-art static taint analysis tools.

1.3.3 Uncover Data Access and Reduce False Negatives of Static Taint Analysis through Machine Learning Techniques

ConDySTA replies on dynamic taint analysis to discover sensitive data access, which is limited on the test coverage, one data access can be missed if the method was not triggered during testing. In order to trigger sensitive data related events as many as possible, ConDySTA requires human effort to register and login to the app, making it not feasible for use on a large scale. To avoid human effort and the limitation on test coverage, we propose, DAISY, which uses machine learning techniques to identify sensitive data access through methods defined by app and third-party libraries, specifically, to identify app's or third-party libraries' methods that return sensitive information.

DAISY is trained on an automatically labelled data set of methods and their calling context. During training, we run all training apps and collects run-time return values of methods being executed. Then all the executed methods can be automatically labeled by testing whether their return values contain planted sensitive data (we can plant sensitive data such as device ID and account email address before running the apps). Some methods can be partially sensitive because they sometimes return sensitive values but do so only under certain context. Instead of just classifying single method, DAISY classifies methods along with calling contexts (called in-context methods). The same method with different calling context can be labelled differently during training and predicted differently during testing.

We consider methods' signatures as natural language sentences to leverage the advances in natural language processing (NLP) and machine learning to classify a method. While a word embedding provides a robust semantic representation of text, it can hardly handle words not seen in the training set, which are common in method signatures with informal abbreviated texts. We handle informal texts by taking advantage of the sub-word embedding feature of FastText [89, 130] framework. We evaluated DAISY on previously unseen apps and DAISY was able to identify sensitive methods in them with high precision. The identified sensitive methods were used as sources in static taint analyses and detected privacy leaks that were missed by existing tools.

1.4 Contribution

In short, this thesis contributes to data in mobile apps including:

- We conducted an empirical study on how apps collect and process user's data through analytic services, which is the most widely used service for analyzing user's data. We studied the real practice of 1,000 popular apps from the Google Play store using analytic services.
- We found 120 out of 1,000 apps share user's PII with analytic service without encryption.
- We manually inspected the policies of the 120 apps and found 27 of them may violate their own privacy policies by sharing PII with third-party services.
- Taint analysis is a widely used technique for detecting sensitive data access and transmission. We proposed a new approach for reducing false negatives of static taint analysis by utilizing dynamic taint analysis to uncover sensitive data access through methods defined by app and third-party services.
- We demonstrated the feasibility of using the uncovered sensitive data access as source to detect more privacy leaks. Among 100 top Android apps from Google Play Store, we detected 39 leaks which are missed by the state-of-the-art static taint analysis tool.
- We proposed a machine learning based approach to identify sensitive methods defined by apps and third-party libraries
- We provided manually and automatically labeled data sets of in-context methods with sensitive information types that can be leveraged in future research.

This dissertation is organized as follows. Chapter 2 introduces the background and related work. Chapter 3 describes our empirical study on how apps collect and process user's data through analytic services. Chapter 4 presents our hybrid taint analysis approach to reduce false negatives in static taint analysis. Chapter 5 describes our machine learning approach for identifying sensitive

methods defined by app and third-party libraries. Chapter 6 concludes this thesis and discuss our future work.

Chapter 2: BACKGROUND AND RELATED WORK

In this chapter, I will summarize the general background of this study and a review of related literature.

2.1 Regulations

2.1.1 Privacy laws and PII

The EU General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA) are two of the most conspicuous privacy protection laws. Both give consumers legal rights in many circumstances to access personal data collected or processed by a regulated entity. Those regulations applies to mobile apps that collect and process personal data.

Personally identifiable information (PII) is the most sensitive data and GDPR has the following definition [29]:

> "Personal Data: an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person."

Based on the definition from GDPR, PII is the information that could be used on its own to directly identify, contact, or precisely locate an individual. This includes: email addresses, mailing addresses, phone numbers, precise locations, full names or usernames. In mobile apps, sharing user's PII with other entity could raise privacy risks.

2.1.2 Privacy policies

A privacy policy serves as an essential way to communicate with users regarding which and how user's information has been accessed, collected, and the purpose of the data collection and

sharing. Users read the policies to make informed decisions on accepting the privacy terms before installing the apps. For example, the following excerpt from the TikTok app's privacy policy listed on Goole Play [73]:

> "**Device Information**
>
> We collect information about the device you use to access the Platform, including your IP address, unique device identifiers, model of your device, your mobile carrier, time zone setting, screen resolution, operating system, app and file names and types, keystroke patterns or rhythms, and platform.
>
> **Location data**
>
> We collect information about your location, including location information based on your SIM card and/or IP address. With your permission, we may also collect Global Positioning System (GPS) data."

Existing works have been working on detecting misalignment between privacy policy and the actual data practice in app code [183, 201, 217, 226]. They analyzed the app code and detected what sensitive information types from user input or Android platform API invocations are sent to network. After that, they compared the collected and shared with information types with the statements in privacy policies.

2.1.3 Terms of service from third-party services

Mobile app developers often share user's data with third-party services to improve their apps, such as user behavior analysis or crash analysis. In order to protect user's privacy and avoid unnecessary data sharing, many services provide documentation specifically discouraging or prohibiting sharing user's PII when using their services. For example, Flurry, a popular analytic service, has the following text in its documentation for their API *setUserID()* [25]:

> "Warning: It is a violation of our terms of service to track personally identifiable information such as a device ID (e.g. Android ID) using this method. If you have a

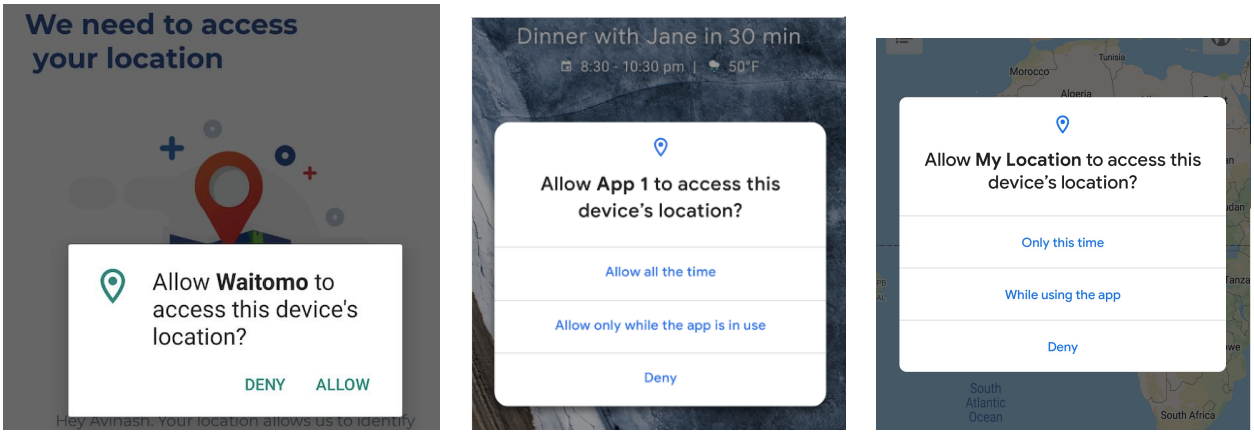
user login that you wish to send to Flurry using this method, you must anonymize the data using a hashing function such as MD5 or SHA256 prior to calling this method."

2.2 User Data Collection and Sharing in Android

2.2.1 Permission and Platform Resources

Android utilizes the Linux security model and layers through a user-based permission system. The Android permission system aims to control the access of mobile apps to sensitive resources. In the latest android platform, there exist more than 200 permissions controlling different resources. Through the permission system, an app can access resources such as the device's GPS location, camera, network connections, and other sensors [223]. To access private user resources, an app needs to declare the corresponding permissions in the manifest files. At the installation time or when the app is performing operations that require permissions during runtime, the user will be asked to grant or refuse the permissions. If the user grants the permissions, the app can access the private user data. If the user refuses the permissions, the app won not have access to the requested resources.

With growing privacy concerns, Android permission system evolved to let the user has more control over their private data. As Figure 2.1a shows, in Android nine and lower, users made persistent choices when granting permission to apps. They could either deny or allow, the latter of which gave apps access all the time (foreground and background). To avoid resource access in the unnecessary scenario, Android 10 gives users three options for allowing the app access to a device's resources. As Figure 2.1b shows, if the user selects "Allow only while the app is in use", an app can access location only while the user is using the app. Starting from Android 11, whenever an app requests a permission related to location, microphone, or camera, the permission dialog contains an option called "Only this time". As Figure 2.1c shows, if the user selects this option in the dialog, the app will be granted a temporary one-time permission.



(a) Location permission in Android 9 and below (b) Location permission in Android 10 (c) Location permission in Android 11

Figure 2.1: Permission system revolution

2.2.2 Android APIs

Android Platform provides large-scale application programming interfaces (APIs) to support application development. These APIs enable the developer to access the system's features and resources such as user data, settings, and hardware. To access the user's data, once the user grants permissions, the app could access the device's corresponding resources through the Android resources APIs. For example, if the LOCATION permission has been granted, the developer could call *getLastLocation ()* to retrieve the device location. If the READ_PHONE_STATE permission has been granted, developers could call *getDeviceId()* to retrieve the device's IMEI.

2.2.3 Data Sharing with Third-Party Services

Mobile app developers often rely on third-party services to improve their apps, such as integrate social network, and crash analysis. Among the most popular types of third-party services, analytic services enable app developers to gather user behavior information to improve their products and monetize their apps with targeted ads. Such analytic services can be integrated into apps through package libraries. App developers use the provided APIs to collect and send user activities to the analytic servers for analysis. The server-side analysis will then generate aggregated reports for the app's developers. Existing research efforts mainly studied what user activities are

tracked by analytic services and what information they may collect. Liu et al. [142] investigated the types of user activities being tracked by analytic services. Their results reveal different levels of user-activity tracking on different UI event types. Since analytic libraries are integrated into the app, they receive the same privilege (e.g., permissions) of the enclosing app from the Android platform. This allows the analytic services to collect some personally sensitive device information. Seneviratne et al. [176] show that 60% of paid apps are connected to analytic services that collect personal data compared to 85% - 95% of free apps. They perform static analysis on Android API calls inside the analytic libraries and summarize the type of personal data collected by the analytic services from the Android platform.

There has been a lot of work on the detection of information leask on mobile platform to third-parties. In particular, CLUEFINDER [152] leverages NLP technology for building a learning system to identify sensitive data leaks from apps to third parties. Network traffic analysis techniques have also been applied to detect personal data that app share with third parties [164] [167] [166]. Razaghpanah et al. [163] detect third-party advertising and analytic services at the traffic level. Ren et al. [167] instrument VPN servers to identify privacy leaks in network traffic. Vallina et al. [195] analyze mobile ISP traffic logs to identify advertisement traffic.

2.3 Taint Analysis

To ensure that information is only used in accordance with the relevant confidentiality policies, it is necessary to analyze how information flows within the using program [174]. There is a large body of work towards enforcing secure information flow through static, dynamic or hybrid program analysis. The common static techniques for enforcing secure information flow including type systems [159, 199], model checking [190] and dataflow analyses [134, 179]. Purely dynamic enforcement performs dynamic security checks during execution. A monitor is often developed to track implicit information flow [182], timeout instructions [173], and information release or declassification [82].Hybrid enforcement combines static and dynamic techniques. It has the advantage of increasing permissiveness because more information about the actual execution trace

is available at runtime, while also minimizing runtime overhead as some static information can be gathered before the execution [98, 135, 136, 141].

Taint analysis is a type of information flow analysis in which objects are tainted and tracked using dataflow analysis. Static taint analysis is performed prior to execution by considering all possible execution paths. It has been used to analyze data lifetime for Android applications [79], web application [192], and exploit code detection [202]. Dynamic taint analysis is more precise than static taint analysis as it only propagates taint along the real path taken at run time. It is used in a variety of security applications, including data flow policy enforcement [153, 160, 212], malware analysis [181] and Android security [109].

For smartphone apps, a data leak occurs when private data (phone numbers, device identifiers, contact data) flows from sensitive sources to public sinks (Internet, SMS transmission). In this case, sensitive data is leaked. Taint analysis is most frequently used to detect such leaks: it taints sensitive data at its source, and propagates the taint information through the application (or even a combination of apps), issuing a warning if tainted data reaches a sink.

2.3.1 Source and Sink Definition

Taint analysis aims to discover connections between sources and sinks. This requires a definition of sources and sinks. When using taint analysis to detect privacy leaks, we are interested in whether user's sensitive data flows to untrusted parties. For example, if an app reads the user's location from his device and send it to a remote server though network. In this example, the source is the method that reads the location data, and the sink is the method that send this data to the remote server. In the code example from Listing 4.1, which is from the DirectLeak1 test case in DroidBench [80], the user's device ID is read and send out as the text of an SMS message. In this sample code, the device ID is accessed though the Android API getDeviceId() (on Line 4), which is the source, and the the device ID is sent out though the method sendTextMessage (on Line 8), which is the sink.

```
1  void onCreate() {
2     // Get the data
3  TelephonyManager mgr = (TelephonyManager) this.getSystemService(TELEPHONY_SERVICE);
4  String deviceId = mgr.getDeviceId();
```

```
5
6   // Leak the data SmsManager sms = SmsManager.getDefault();
7  sms.sendTextMessage("+49 1234", null , deviceId , null , null);
8  }
```

Listing 2.1: Simple Data Leakage Example.

To perform taint analysis, these source and sink methods need to be defined as input. Android provides certain APIs for the app to access data from the operating system, such as the GPS location, or device identifiers. Previous research attempted to identify APIs that return sensitive information or can be used to send data to third parties. Many approaches [161, 183] focus on platform information extracted from the Android device through the Android APIs. Consider that the Android operating system contains a very large number of public API methods, it is impossible to manually verify each of them. Therefore, Rasthofer et al. proposed Susi [161], a tool to automatically identify source and sink methods in the Android API. It utilizes machine learning to take a small set of manually annotated source and sink methods as their training set. More recent work [126, 151, 201] further consider user's input data as source, which cannot be identified using the Android API alone, and requires tracing potentially sensitive data through GUI API method executions, e.g., `android.widget.EditText.getText()`. This tracing requires classifying the information types by first analyzing the GUI hierarchy [171] and then classifying labels associated with the method invocations.

2.3.2 Taint Analyses for Android

Taint analysis [78, 108] can detect taint flows in software programs and has a wide range of applications in privacy leak detection [90, 126, 151, 201]. FLOWDROID [79] leverages static taint analysis with tunable sensitivity to trace information from sources to sinks so it can also be used to detect information leaks. Other Android-oriented static information analysis techniques include CHEX [145], LeakMiner [213], and ScanDroid [115]. Specifically, CHEX [145] detects component hijacking vulnerabilities in Android applications by tracking taints between externally accessible interfaces and sensitive sources or sinks. LeakMiner [213] is an earlier context-insensitive information-flow analysis technique for detecting privacy leaks in Android apps. ScanDroid [115]

tracks taint flows among multiple apps and detects privacy leaks into other apps.

There are also dynamic taint analysis techniques such as TaintDroid [108] and CopperDroid [189] that perform OS-level or application-level of dynamic taint propagation. TaintArt [187] and TaintMan [215] further extends the existing dynamic taint analysis to support Android RunTime (ART) which adopts ahead-of-time compilation strategy and replaces previous virtual-machine-based Dalvik. Jung et al., [132] proposed PrivacyOracle, which uses differential analysis of tainted values perform dynamic taint analysis on black-box systems without instrumenting the application or the underlying OS. Tripp et al. [193] utilized Bayesian reasoning to determine if an information release at a sink point represents a privacy leak. It calculates the possibility of legitimate information releases at a sink based on the distance between the information about to be released and the original sensitive data. Continella et al. [101] proposed a black-box analysis tool to detect privacy leaks in mobile apps by analyzing network traffic.

2.3.3 Tackling Practical Unsoundness of Static Analysis

Prior researchers have already noticed the unsoundness of static analysis in practice. Researchers from Coverity [13] explained the challenges of applying static analysis to real world [86], and they mentioned in the paper that the static inaccessibility to code as one of the major challenges. In academia, different dynamic supplements of static analysis have been proposed. On handling reflections, Livshits et al. [144] proposed an approach to statically infer information about reflective call sites from program code. TAMIFLEX [88] perform dynamic analysis to record destinations of reflection calls and use such records to supplement the program call graph, which is the basis for many static analyses. [191] tests while instrumenting reflection methods to record the classes constructed from invocations of the reflection methods. Then, for each invocation of a reflection method, it adds an reflective edge from the class that invoked the reflection method to the class constructed by the reflection method. [224] reveal the program behaviors caused by dynamic code update techniques, such as dynamic class loading and reflection. It uses a modified Android virtual machine to log all triggering actions of reflective calls. DroidRA [139] uses static

constant propagation to estimate potential reflection call destinations in Android apps. On handling dynamically loaded / generated code, Wei and Ryder [206] developed blended taint analysis for JavaScript which summarizes dynamically generated code from dynamic analysis output and perform static taint analysis based on the summaries. AVERROES [76] generates mock libraries with analysis summaries so it can be used for replacement of missing libraries. Dufour [107] proposed to collect calling structure data at run time, and feed it as input to static method-escape analysis, so that some complicated code portions can be analyzed more efficiently. PRuby [116] by Furr et al. is a static-type inference system for the Ruby programming language. It uses dynamic profiles to handle the three dynamic language features in Ruby: send, require and eval, which performs reflection invocations, dynamic code loading, and dynamic code generation, respectively.

2.4 Assistant Taint Analysis by Machine Learning on Code

To identify sources and sinks in a program, there have been techniques on the classification of elementary code units such as methods and variables. SUSI [161] uses machine learning to identify Android platform API methods that retrieves sensitive information. Many works use machine learning to perform certain tasks on code. CODE2VEC [77] trains code embeddings to infer high level semantics of code, and their evaluation is performed via method name prediction, where they generate method names from the method code. Vasilescu et al. [196] proposed a machine learning based approach to infer variable names from obfuscated code. Hellendoorn et al. [121] examined the feasibility of using deep learning on source code. They compare deep learning models with statistical models and mixed models (i.e., a combination of statistical and deep learning models) over performance on source code word prediction. Hindle et al. [122] showed the similarity exists between code and natural language, and code is even less surprising than text. It implied that NLP technologies can also be applied to deal with code. Hou et al. use a combination of deep learning and dynamic analysis to detect malware in Android apps. Their system, DEEP4MALDROID, uses a dynamic analysis approach called Component Traversal to maximize the code executed for a given app [125]. A deep learning framework is then applied over the resulting graph to identify

malware based on Android system calls. While deep learning and program analysis are used by DEEP4MALDROID to detect security-related constructs, the system entails a complete, or nearly complete, directed graph of the app in question, rather than an individual, arbitrary method. Xie et al. use dynamic analysis to detect anomalous runtime behavior in high-performance computing systems [211]. The approach builds tree representations (*CSTrees*) as feature vectors from the stacks and applies a One-Class Support Vector Machine to detect anomalies.

Chapter 3: DATA COLLECTION AND SHARING: A STUDY ON ANALYTIC SERVICES

Before analyzing sensitive data flow in mobile apps. It is necessary to gain a deeper understanding of mobile apps with regard to their way to collect and share user's privacy-related data. In this chapter, I will provide technical details of how we obtained metadata and binary files (also known as apk files) of apps from Goole Play, and how we decompile and instrument these apps to observe data collection and sharing with third-party services.

3.1 Overview

Mobile apps often rely on third-party services to enhance user experience through features such as social network integration and crash analysis. Among the most popular types of third-party services, *analytic services* enable app developers to gather user behavior information to improve their products and monetize their apps with targeted ads. Such analytic services can be integrated into apps through package libraries to collect user activities and send user behavior to their servers for analysis. Server-side analysis can then generate aggregated reports for the app's developers. For example, such aggregated reports may describe how many users are from New York City, how many users reached a specific activity, or how long they tend to spend on a specific activity.

Analytic services provide specific methods that allow app developers to set attributes for their users, we refer to those methods as *Attributes Setting Methods (ASMs)*. For example, one commonly used category of ASMs is "set user identifier", which allows app developers to store a user ID for the individual using their apps. These methods are usually optional and can be used to recognize the same user across multiple usages of an app. Once a unique ID is assigned through such a method, the user's behavior reports will be labeled with the provided user ID. These identifiers are strictly used for identification with respect to the service and do not need to be *personally* identifying. For example, a random, unique number or hash value could be used instead of an

email address. Using personally identifiable information (PII)[1] as an ID would be considered as bad practice in this case as it presents an unnecessary exposure of sensitive data. By misusing PII (e.g., email, username, device ID) with ASMs this effectively un-anonymizes the reports produced by the analytics service resulting in privacy risk. Furthermore, such misuse may violate the app's own privacy policy, the analytic service providers' terms of service, or general best practices (e.g., data overuse, least privilege).

A major privacy risk associated with third-party analytic services is the data usage after the behavioral reports have been collected by the analytic service. Once the data have left the app and reached an analytic service, the developers and users lose control of the information. Even if the third-party service is trusted not to misuse the data, accumulated long-term storage of unanonymized user behavioral data is susceptible to theft or leakage [8, 27, 38]. Not expecting PII to exist in the collected behavioral reports, third-party services may share the data to their business partners or do not provide enough protection for them. Furthermore, when multiple apps use the same PII for the same analytic service, multiple behavioral reports can be combined to build more comprehensive personal profiles.

Legal requirements such as EU General Data Protection Regulation (GDPR) requires lawful basis (e.g. legal obligation, explicit consent) to process users' data [30], unless the data is anonymized [28]. For these reasons, it is imperative that unnecessary use of PII for behavioral-report labeling to be eliminated.

Many of the most commonly used analytic services provide documentation specifically discouraging or prohibiting the use of PII as user attributes when using their ASMs. For example, Google's Firebase [11] includes the following in their documentation [24] for configuration of ASM `setUserProperty()`:

> *"When you set user properties, be sure to never include personally identifiable information such as names, social security numbers, or email addresses, even in hashed form."*

Flurry, another popular analytics service, has the following text in its documentation [25] for ASM

[1] We use the union of GDPR and Google Analytics definitions for PII [29, 66].

`setUserID():`

> *"**Warning**: It is a violation of our terms of service to track personally identifiable information such as a device ID (e.g. Android ID) using this method. If you have a user login that you wish to send to Flurry using this method, you must anonymize the data using a hashing function such as MD5 or SHA256 prior to calling this method."*

App developers may also attempt to reduce PII-related misconfigurations by adopting privacy policies that require anonymization or aggregation of data when used with analytic services. For example, the privacy policy for the app ShopClues [64] claims:

> *"ShopClues.com may also aggregate (gather up data across all user accounts) personally identifiable information and disclose such information in a non-personally identifiable manner to advertisers and other third parties for other marketing and promotional purposes."*

Despite such documents and policies, it is not clear whether app developers always follow them in reality as they may neglect them during development. In this paper, we perform a study to understand how app developers invoke ASMs in practice and whether those practices comply with the documents and policies of the analytic service providers and the apps themselves. It should be noted that, while there exist research efforts on data collection behavior, over-privilege, and leak detection for third-party libraries [127, 164, 167, 176, 178], our work is different in that it studies the cause of leaks related to misconfiguration of third-party services. Specifically, we try to answer the following four research questions in this study.

- **RQ1:** What configuration methods do analytic service provide and how do apps invoke those methods?
- **RQ2:** How commonly do app developers use PII when configuring analytic service?
- **RQ3:** Do analytic services provide mechanisms to protect anonymity in the case of misconfiguration as a result of **RQ2**?

- **RQ4:** Do analytic service misconfigurations result in violations of apps' own privacy policies and analytic service providers' documents/policies?

To answer these research questions we developed a semi-automatic approach, Privacy-Aware Analytics Misconfiguration Detector for Android (PAMDROID), to detect and analyze misconfigurations that may lead to privacy risk. In this approach, we first investigated the documentation of the 18 most popular analytic services in the mobile analytic ecosystem as listed in AppBrain [10]. We acquired the methods provided by these analytic services through their Application Program Interface (API) specifically for configuring user attributes (ASMs). We also collected the configuration instructions and terms of service notices from these analytic services, when available, to gather their guidelines and recommendations for use. With this data, we designed and conducted an experiment to dynamically and automatically evaluate the top 1,000 Google Play store apps that contained at least one ASM invocation in their code. We detected invocations to attribute-setting ASMs at run time and recorded parameter values to study what the common practices were and whether they abode by the app's privacy policies, the analytic service guidelines, and best practices concerning PII for using analytic services. We also investigated the analytic reports generated by the analytic services to study whether the services applied any mechanisms to anonymize or aggregate the collected data.

We have the following major findings:

- Based on the results of our semi-automated approach, 555 out of 1,000 top apps from the Google Play store had at least one ASM invocation observed at run time and 120 of them used PII to configure analytic service without encryption.

- All the analytic services we investigated provide behavior reports on individual users to app developers and the reports are labeled with exactly the same identifiers provided by app developers. Therefore, if PII is used as an identifier, they will be directly linked to the user behavior reports, resulting in targeted, non-anonymous and non-aggregated information.

- We manually inspected the policies of the 120 apps and found 27 of them may violate their

own privacy policies by using PII as user attributes.

- Using PII with analytic services may also violate the Terms of Service (TOS) of analytic services. Among the analytic services we studied, we found that four of them explicitly require app developers to avoid passing PII to ASMs. They are Firebase, Google Analytics, Flurrry and Mixpanel, and they have the app-market shares of 55.95% [57], 26.84% [59], 5.12% [58], 0.77% [60], respectively. Although only four analytic services state this requirement explicitly, Firebase, Google Analytics, and Flurry are the top three market share holders and dominate the market, so we believe this requirement is a standard for analytic services. Our result shows that 37 apps which are using the four analytic services did set user's PII to the ASMs, and thus may violate analytic services' terms of service (Firebase, Google Analytics, Flurrry) or privacy guidelines (Mixpanel).

3.2 Analytic Services

For a better understanding of users' behavior, app developers often choose to utilize analytic services. Analytic services usually provide client libraries that app developers could utilize in their app, which will record an app user's interaction with the app and send the corresponding data to the server of the analytic service. Later, the analytic services can link the activity of a mobile app user over time into a behavior report. The behavior report includes detailed usage information about this user. The analytic services can then aggregate all the users' reports and provide analytic data to the app developers so that they can improve their product or make better business decisions based on the analytic report.

3.2.1 Tracked User events

Analytic services automatically collect some events that are triggered by basic interactions such as ad impressions, ad clicks, and screen transitions. Table 3.1 shows the default events collected by Firebase and Mixpanel. From the table, we can see that the collected events contain detailed information about the user's usage of the app and interactions with the ads.

Table 3.1: Analytic services collect user events by default

Firebase	ad_click, ad_exposure, ad_impression, screen_view, user_engagement, session_start, app_clear_data, app_exception, etc. [18]
Mixpanel	first app open,app updated, app crashed, app session in app purchase. [40]

3.2.2 Analytic Service Configuration

Analytic services provide *Attributes Setting Methods (ASMs)* that enable developers to customize the analytic service by setting some attributes for their users. Developers can set identifiers or other attributes such as age, gender, and location on each app user. Later, developers can use those attributes as a filter or metrics in their analytics reports. For instance, a developer may want to know the geography distribution, or age distribution of their users. The data that developers pass to those ASMs will be associated with the users' collected events and then sent to the server of analytic services. To protect users' privacy, analytic services have certain guidelines or suggestions for how the developer should use those ASMs. We list two from some analytic services here as examples:

In Firebase [24] [22]:

> When you set user properties, be sure to never include personally identifiable information such as names, social security numbers, or email addresses, even in hashed form.
>
> Note: You are responsible for ensuring that your use of the user ID is in accordance with the Google Analytics for Firebase Terms of Service. ... For example, you cannot use a user's email address or social security number as a user ID.

In Mixpanel [42]:

> If you wish to track users truly anonymously, however, then your tracking implementation should not use user-specific information, such as the user's email address. Instead use a value that is not directly tied to a user's PI (personal information), whether it be a unique anonymous hash, or a non-PI internal user identifier.

These instructions require the app developers to not use any PII to configure analytic services and encourage them to use anonymous data instead.

3.2.3 Personally Identifiable Information

We consider PII as the union of the definitions by Google Analytics and the EU General Data Protection Regulation (GDPR). The following statement is from Google Analytics [66].

> "*Google interprets PII as information that could be used on its own to directly identify, contact, or precisely locate an individual. This* ***includes****: email addresses, mailing addresses, phone numbers, precise locations (such as GPS coordinates - but see the note below), full names or usernames*"

The following statement is from GDPR [29].

> "***Personal Data:*** *... an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person.*"

GDPR also defines online identifiers [31] which we include as PII:

> "***Online Identifiers:*** *Natural persons may be associated with online identifiers provided by users' devices, application, tools or other identification tag and it could be used to associate with natural persons, because online identifiers may leave traces which, in particular when combined with unique identifiers and other information received by the servers, may be used to create profiles of the natural persons and identify them.*"

3.3 PAMDroid and Study Design

The goal of this research is to detect misconfigurations in analytics services as they may lead to privacy risks. To this end, we developed PAMDroid, a semi-automated approach to detect

the misconfiguration of analytic services due to setting PII to ASMs. As illustrated in Figure 3.1, there are two manual preparation steps of PAMDROID. First, we manually collect a set of most popular analytic services and Android apps. For each analytic service, we investigate its API documentation to collect all ASMs that app developers can use to set user attributes. Second, we set up an Android device and collected all its information to construct a *reference user profile*. The profile includes different platform IDs (e.g., device ID, serial number, Android ID, advertising ID), a synchronized Google account (e.g., user name, user email, address, age, gender, date of birth), and other sensitive information (e.g., location, IP address, MAC address).

After these two steps, PAMDROID first performs static smali code analysis on the apps to filter out the apps that do not invoke any ASMs at all. Then, PAMDROID automatically instruments all ASMs (detected with static smali code analysis) to print their argument values to system log. After that, PAMDROID uses Monkey [65] to test the instrumented apps' user interface. Note that many apps trigger analytic services only after a user is logged in. As a supplement of Monkey, we perform manual login for all apps that require login to get to the start page. Finally, PAMDROID compares the collected system logs with the reference user profile. When certain types of information in the reference user profile show up in the system log, PAMDROID detects an ASM misconfiguration. After all misconfigurations are detected, we manually inspect the corresponding apps' privacy policies and corresponding analytic services' terms of services to detect violations and misalignments. It should be noted that the major goal of this research is to study the commonality and characteristics of ASM misconfigurations, and PAMDROID is developed for the study, so we supplemented it with manual analysis to acquire most comprehensive and accurate results. If we do not perform manual log-in and adopt existing automatic approaches for policy analyses [183, 201, 217, 226], PAMDROID can be made fully automatic, but its effectiveness is not clear and it is not the focus of this paper.

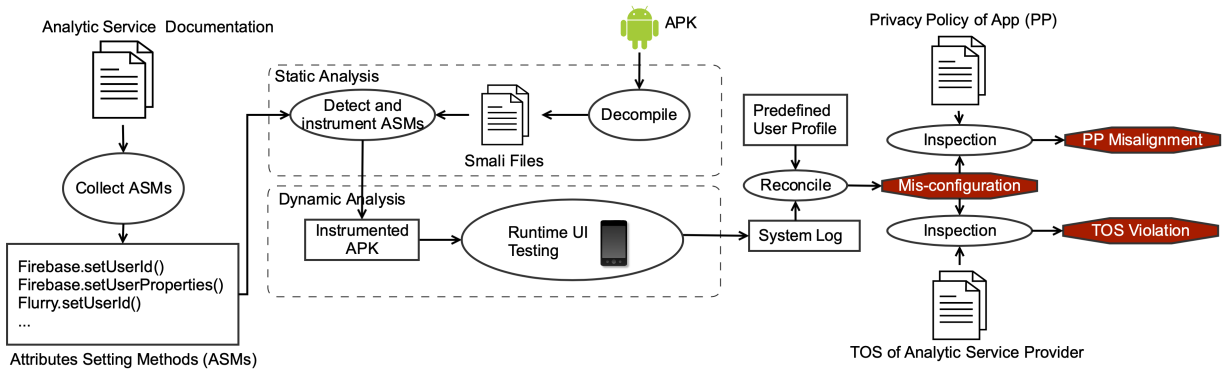


Figure 3.1: Privacy-Aware Analytics Misconfiguration Detector(PAMDROID)

3.3.1 Collection of Apps and Analytic Services

We identified the 18 most popular analytic services using published statistics provided by AppBrain [10], a company specializing in app marketing and promotion. After that, we identified the ASMs provided by the selected analytic services. The top 1,000 free apps containing at least one invocation of the studied ASMs were collected from PlayDrone [45], a collection of metadata for Android apps on the Google Play store. We identified those apps which invoked ASMs by analyzing their smali code[2]. If an app obfuscated the ASMs it invoked, we could not apply our approach to it. Furthermore, we also ruled out apps that were incompatible with our device and those no longer existing in Google Play due to being removed since being included in the PlayDrone database.

To determine whether an app had an invocation of a studied ASM, we first decoded the analytic libraries into smali format using APKTool [194] and identified each ASM's smali signature. We then decompiled each app's APK (Android Package) file into smali format and scanned the resulting file for occurrences of ASM signatures. Only apps containing at least one ASM signature were kept for consideration.

[2]Assembler for the dex format used by Dalvik

3.3.2 Runtime Information Collection

There are multiple approaches to detect information flow to ASMs. The first approach we considered was using static taint analysis. To this end, we used FLOWDROID [79] to analyze the 1,000 apps and defined ASMs as sinks and personal information sources from SUSI [161] as sources. The result showed that FLOWDROID only identified 10 data flows from sources to sinks. Through further investigation, we found that the sources of PII sent to ASMs are often not Android API methods, but system files or databases. Furthermore, PII often flow through paths that are not handled by FLOWDROID, such as `android.content.SharedPreferences`, which is a data structure in Android system that stores user information such as username, device ID, etc. If we add all these API methods as sources of FLOWDROID, it will report many false positives as files, databases, and Android system data structures may also contain a lot of non-PII.

To make sure our study is conservative (all reported misconfigurations are real), we ultimately utilized value-based dynamic taint analysis. As mentioned earlier, we prepared a reference user profile to match arguments sent to ASMs. To make sure values in our user profile are not confused with other values, we designed very strange information (e.g. user name, email address) for the synchronized Google account. To make sure our matching is robust, for the values in the reference user profile, we further generate values with different value transformations, such as reverting and truncating. We also produced hashes for all PII using common hashing algorithms provided by Android API methods so that we could identify hashed values (although in the study we did not find hashes being sent). Note that we manually confirmed all matched results to make sure that our value-transformations do not lead to wrong matches. One limitation of value-based taint analysis is that we cannot detect encrypted PII with an app-specific key. Notably, using encrypted PII as user attributes on analytic service already reduces the risk to privacy, because the unencrypted PII will not be combined with collected user behavior.

In order to catch the arguments of ASM invocations during runtime, we instrumented all ASMs in smali code by adding a call to the Android logger to report the invocation at the beginning of the ASM implementation. This allows us to use the Android system log to analyze method argument

Table 3.2: System log of ASM invocation

1	01-10 18:16:55.024 1931 1931 W System.err: java.lang.Exception: Third-party API invoke detection:Print StackTrace with parameter:
2	01-10 18:16:55.024 1931 1931 W System.err: at com.google.firebase.analytics.FirebaseAnalytics.setUserProperty(Unknown Source)
3	01-10 18:16:55.024 1931 1931 W System.err: at com.vivino.android.a.a.a(FirebaseHelper.java:160)
...	
...	
17	01-10 18:16:55.024 1931 1931 I Third-party API invoke detection:Print StackTrace with parameter:: vivino_email
18	01-10 18:16:55.024 1931 1931 I Third-party API invoke detection:Print StackTrace with parameter:: *******@gmail.com

values being set at runtime. After inserting the code, we rebuilt the smali code back into APK format for testing. We used the Android Debug Bridge (adb) to automatically install the rebuilt apps onto our test device and run the apps and then executed Monkey to perform the testing. For each app, we automatically installed, executed, tested, uninstalled and saved the system log into the local file system for later inspection. During testing, we found 254 apps requiring login to an account to show the app's start page, so we manually created accounts for these apps using the reference user profile to complete the login process.

Finally, PAMDROID searched the system logs generated during testing and extracts argument values of ASMs based on flags inserted during instrumentation. Table 3.2 is an example where Line 1 shows our inserted flag; Line 2 shows the ASM that be invoked (`Firebase.setUserProperty`), and Line 17 shows our flag and the first argument value that was passed to the ASM ("vivino_email"). Line 18 shows the second argument value which was the email address (represented as "********@gmail.com").

3.4 Study Results

3.4.1 Apps' Usage of Analytic Services

To answer **RQ1**, for each analytic service, we first investigated their documentation and collected the ASMs. We noticed that every analytic service provides the methods that allow developers to set attributes for their users, such as `setUserID`, `setCustomerUserId`, or `setUserIdentifier`, etc. Firebase provides an method called `setUserProperty`, which allows developer to set any attributes to describe their user. It takes two arguments which are similar as a pair of "key" and "value". Other methods include `setUserEmail`, `setLocation`, `setAge`, `setGender`,

`setDeviceId`, `setPhoneNumber`, etc. The full list of ASMs are available at our anonymous project website [3]. Four analytic services (Firebase, Google Analytics, Flurry, Mixpanel) explicitly require app developers to avoid setting PII [22, 23, 25, 42] to ASMs.

A method to set user identifier (e.g., `setUserId`) is provided by every analytic service and mostly commonly invoked in our test. For example, `Crashlytics.setUserIdentifier` was invoked in 147 apps, and `Flurry.setUserId` was invoked in 67 apps. We present these frequencies in Table 3.3. In the table, the first column presents the analytic service name; the second column presents the total number of apps that invoked the ASMs from this analytic service. The third column represents the ASM name; and the fourth column presents the number of apps that invoked the corresponding ASM. Among the 1000 apps that contain ASM invocations in their smali code, 555 apps invoked 29 ASMs from 13 different analytic services during our runtime testing. Table 3.3 shows that Firebase and Crashlytics are the most commonly invoked analytic services. Note that a single app could use more than one analytic services, within one analytic service, the app could invoke multiple ASMs to set user attributes.

To understand how apps use different types of ASMs over all analytic services, we categorized all ASMs in Table 3.3 into a number of categories according to their purposes. In particular, the categories are "set user identifier", "set user properties", "set device identifier", "set user email", "set username", "set age", and "set location". In Figure 3.2, we present the number of apps that invoke different categories of ASMs. We observed that 387 apps set user identifiers to at least one analytic service, showing that many app developers set identifiers for users to differentiate individual user interactions through the analytic service, and the function is also well supported by analytic services in general. Furthermore, 198 apps set user properties to at least one analytic service. Since ASMs in the "set user properties" category are very general and can be used to set almost any data, it is difficult to statically tell what information is sent through them.

[3] https://sites.google.com/site/trackersec2019/

Finding 1. Our answer to **RQ1** is that, all our studied analytic services provide ASMs for app developers to set user attributes, and more than half (555 of 1,000) of apps trigger ASMs to label user behavior reports.

Table 3.3: #Apps Invoke Different ASMs

Analytic library	# Apps	Method	# Apps
Firebase	216	setUserId	64
		setUserProperty	193
Crashlytics	163	setUserEmail	21
		setUserIdentifier	147
		setUserName	38
AppsFlyer	81	setAndroidIdData	31
		setAppUserId	4
		setCustomerUserId	63
Flurry	70	setAge	6
		setLocation	2
		setUserId	67
Tune	38	setAndroidId	12
		setDeviceId	3
		setUserEmail	2
		setUserName	2
		setUserId	27
		setFacebookUserId	6
		setGoogleUserId	6
		setTwitterUserId	6
IronSource	24	setUserID	24
mixpanel	17	identify	17
Applovin	13	setUserIdentifier	13
Leanplum	12	setDeviceId	6
		setUserId	4
		setUserAttributes	5
Branch	11	setIdentity	11
Google Analytics	7	setClientID	7
Appsee	6	setUserId	6
Newrelic	4	setUserId	4

3.4.2 PII set to ASMs in Misconfiguration

To answer **RQ2**, we further studied what types of data are set to ASMs in our studied apps. By matching the logged method arguments to the controlled user profiles (see Section 3.3.2), we can detect misconfigurations on the fly.

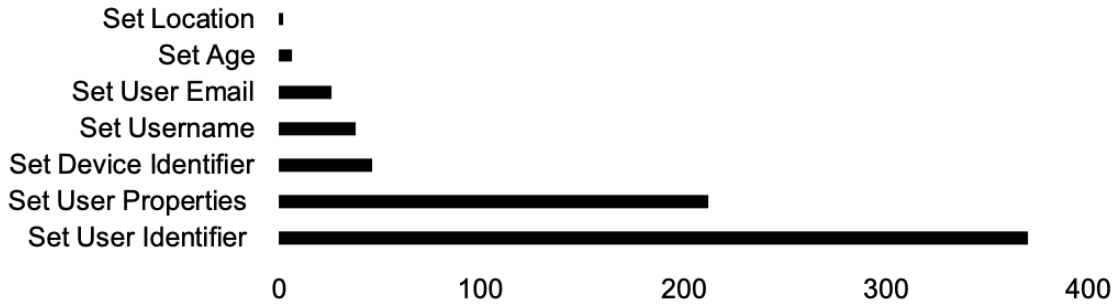


Figure 3.2: #Apps Invoking Different Types of ASMs

Table 3.4 presents the number of apps setting different types of PII to ASMs. In particular, Columns 1-4 present the type of PII, ASM name, the number of Apps setting certain type of PII to a certain ASM, and the total number of Apps setting certain type of PII to all ASMs. We make three major observations. First, overall 120 apps set PII or PII's transformation (11 apps) to ASMs. It should be noted that a single app may set multiple data types, so the values in Column 4 do not add up to 120. Second, among the 120 apps, 79 apps set Android ID to ASMs, 24 apps set users' email addresses to ASMs, and 19 apps set users' registered username to ASMs. Note that registered usernames are used to uniquely identify users in the app, and many users use the same username across apps, so Google Analytics explicitly lists username as PII [56]. Third, one type of PII is observed to be set to ASMs for multiple purposes. For example, Android IDs are mainly set to ASMs in the category "set user identifier", but it is also set to `Crashlytics.setUserName()` and `Firebase.serUserProperty()`. Email addresses are also set to ASMs in the categories of "set user properties" and "set user identifier". So the vagueness and generality of ASM design may have aggravated their misuse.

In Figure 3.3, we further show the number of apps that set different PII to different analytic services. In the figure, we organize the number of apps setting various PII to each analytic service as a separate column chart. In each sub-column-chart, the x-axis shows different analytic services, and the y-axis shows the number of apps setting different personal information type in that analytic service. From the figure, we can see that Crashlytics and AppFlyer are receiving PII from the most number of apps, and Crashlytics also received user email addresses from the most number of apps.

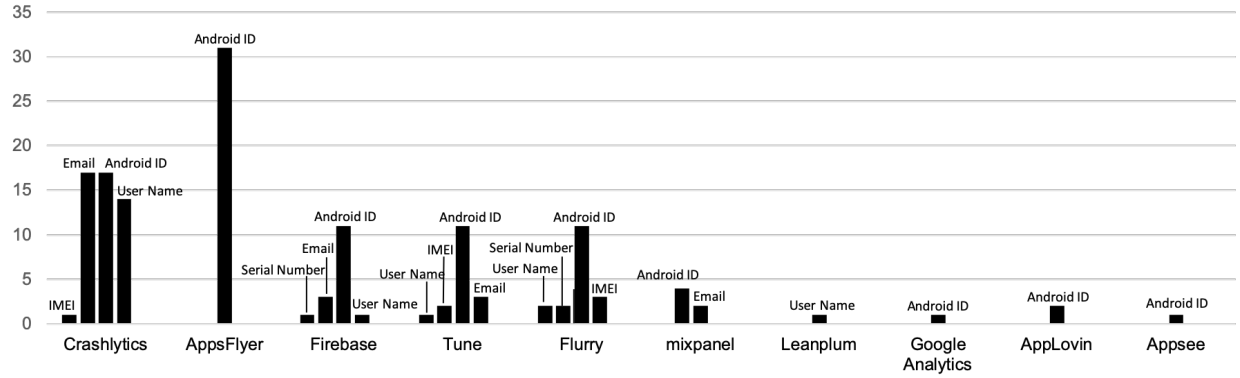


Figure 3.3: # Apps set different PII to different analytic services

Furthermore, Firebase and Flurry, which explicitly require app developers to not send PII to them, both receive various types of PII, including Android ID, device series number, and username. Firebase further receives email address, and Flurry further receives IMEI.

Finally, Figure 3.4 presents the category distribution of our dataset and the percentage of apps (in each category) setting PII to ASMs. Each bar represents the total number of apps in the specific category, while the dark portions represent the number of apps in the category that set PII to the ASMs. We further label the percentage of dark bar portion for each bar. The figure shows that there is not a specific category of apps that are much more likely to use PII as user attributes. Compared with others, apps in Photography, Communication and Shopping have higher possibility of setting PII to ASMs. Besides PII, our test result shows that 24 apps used Advertising IDs, which can be changed by users and sometimes encouraged by analytic services to be used as user identifiers. However, if users do not change Advertising IDs frequently, they can still be actually PII. Since we want our study results to be conservative, we do not include them as PII in our study results.

Finding 2. Our answer to **RQ2** is that, among the 1,000 apps we studied, at least 120 apps (detected by PAMDROID) misconfigure ASMs with PII. In particular, Android ID (in 79 apps), User Email (in 24 apps), Username (in 19 apps), IMEI (in 6 apps), and Serial Number (in 3 apps) are the types of PII being set to ASMs.

Table 3.4: #Apps Setting Different PII to ASMs

Personal Info	Tracker API	#Apps	Total
Android ID	Firebase.setUserId	8	79
	Firebase.serUserProperty	5	
	AppsFlyer.setAndroidIdData	29	
	AppsFlyer.setCustomerUserId	2	
	Flurry.setUserId	11	
	Mixpanel.identify	4	
	Tune.setAndroidId	11	
	Tune.setDeviceId	1	
	Crashlytics.setUserIdentifier	16	
	Crashlytics.setUserEmail	1	
	Crashlytics.setUserName	1	
	Applovin.setUserIdentifier	2	
	GoogleAnalytic.setClientId	1	
	Appsee.setUserId	1	
Email	Firebase.serUserProperty	3	24
	Mixpanel.identify	2	
	Tune.setUserEmail	2	
	Tune.setUserName	1	
	Crashlytics.setUserEmail	12	
	Crashlytics.setUserIdentifier	1	
	Crashlytics.setUserName	5	
Username	Firebase.serUserProperty	1	19
	Flurry.setUserId	2	
	Tune.setUserName	1	
	Crashlytics.setUserName	14	
	Crashlytics.setUserIdentifier	1	
	Leanplum.setUserAttributes	1	
	Leanplum.setUserId	1	
IMEI	Flurry.setUserId	3	6
	Tune.setDeviceId	2	
	Crashlytics.setUserIdentifier	1	
Serial Number	Flurry.setUserId	2	3
	Firebase.serUserProperty	1	

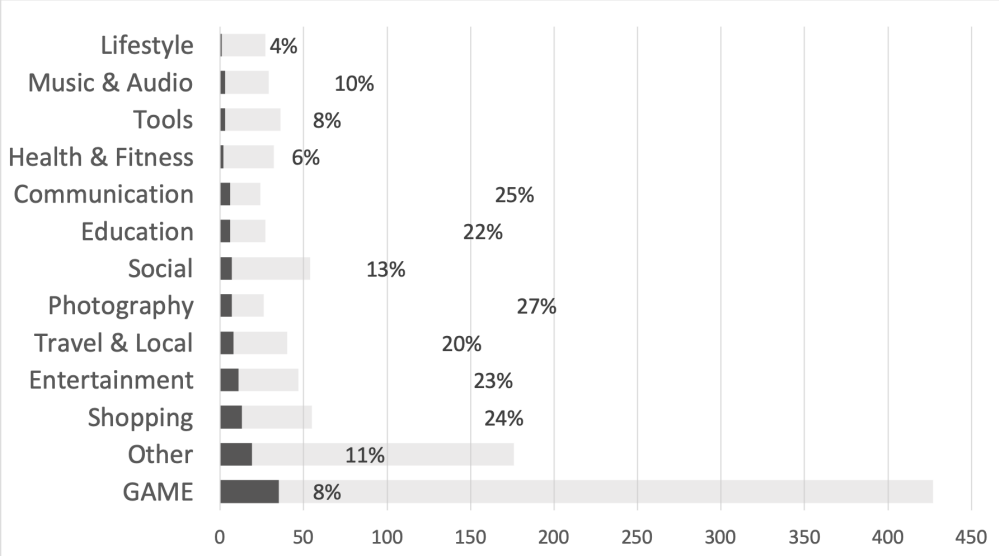


Figure 3.4: # Apps distribution in categories

3.4.3 Enforcement of Aggregated and Anonymous Reports

To answer **RQ3**, we studied all 13 analytic services being invoked to find out whether they have enforcement mechanisms to reject PII being set to ASMs. Unfortunately, none of 13 services have such built-in enforcement mechanisms. Only one of them, Appsflyer [12], provides a method to set user email address with encryption, but none of apps in our data set actually invoked this method. Furthermore, we studied whether the information set to ASMs is encrypted before they are combined with behavior reports, and no analytic service is performing the encryption. It should be noted that all the analytic services which we studied use encrypted network connection (e.g., HTTPS) to send collected information. However, if the PII set to the ASMs is combined with behavior reports in un-encrypted form, the anonymity of the collected user behavior is already lost as the whole data will be decrypted later.

It is very challenging to tell how data is stored and processed on servers of analytic services. However, we can predict their practice from the behavior reports they provided to developers. Therefore, we further studied whether the analytic services provide reports on individual user behaviors. We found that for all analytic services that we investigated, their online analysis reports for

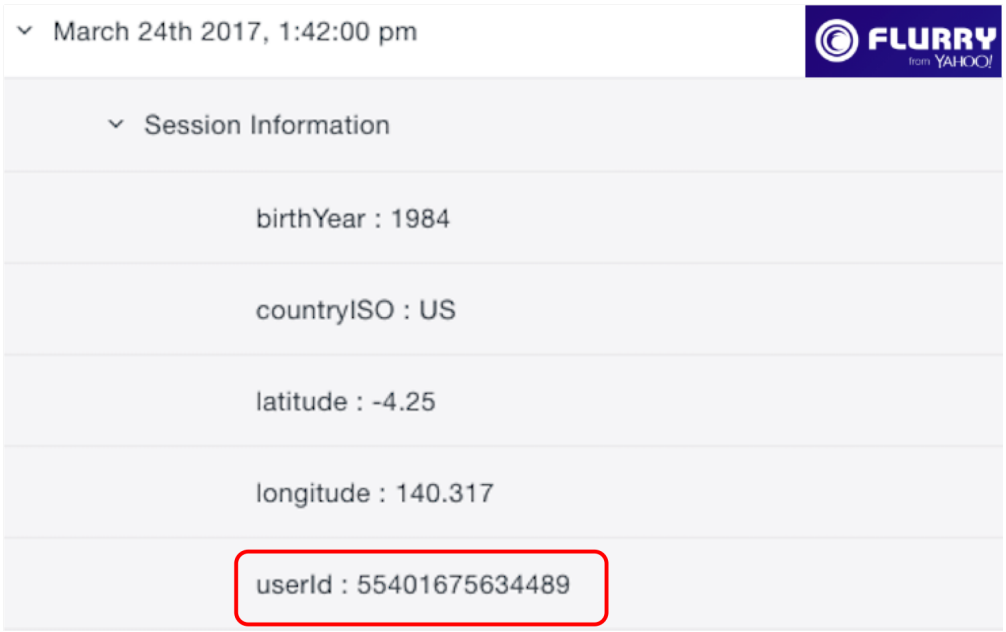


Figure 3.5: A demo report in Dashboard of Flurry [26]

developers are not limited to aggregated data, but are instead itemized by received user attributes. Figure 3.5 , Figure 3.6 and Figure 3.7 presents example report screen-shots from Flurry, Mixpanel, and Crashlytics. From the three figures, we see that reports are organized by user attributes and presented to app developers, and the identifiers (e.g., user email, username or device IDs) are presented without anonymization. Figure 3.5 shows that Flurry's report not just contains the userId, but also user's latitude and longitude data.

Finding 3. Our answer to **RQ3** is that, analytic services do not have any mechanisms to vet or anonymize PII they received from ASMs. The PII are directly combined with behavior reports when stored and provided to app developers.

3.4.4 Policy Violations and Misalignment

We present our answer to **RQ4** in this subsection. As we discussed in our results above, it is a privacy risk when PII was set by app developers on analytic services without encryption or anonymization. Such misconfiguration may cause two types of policy-related issues. First, to protect user privacy and avoid legal liabilities, analytic services may state in their TOS that they

Figure 3.6: A demo report in Dashboard of Mixpanel [105]

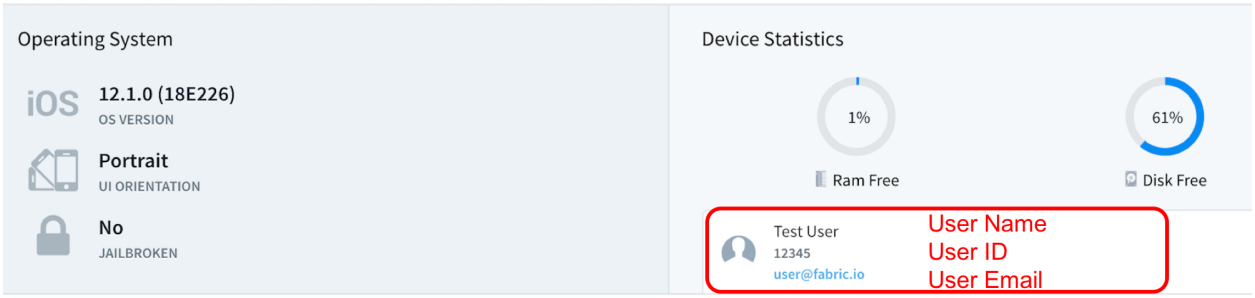


Figure 3.7: A demo report in Dashboard of Crashlytics [15]

do not allow developers to set PII to their ASMs. So the misconfiguration of ASMs will cause TOS violations. Second, the app's own privacy policy may claim anonymous data analytics or fail to describe the sharing of PII to analytic services, so the misconfiguration of ASMs will cause misalignment between code and privacy policies.

TOS Violations

Figure 3.3 shows that 120 apps set PII on ten analytic services. As we mentioned in Section 1, four analytic services (Firebase, Google Analytics, Flurry, Mixpanel) explicitly require app developers to avoid setting PII to ASMs in their terms of service (Firebase, Google Analytics, Flurry, note that they are the top three market-share holders in analytic services) or privacy guidance (Mixpanel).

Based on our experiment results, 31 apps have set PII to ASMs of Firebase, Google Analytic, or Flurry, and thus we believe that the misconfiguration of ASMs actually violates their terms of services. Furthermore, 6 apps have set PII to ASMs of Mixpanel, so they are violating Mixpanel's

privacy guidance. It should be noted that, although the remaining 83 apps did not violate the policies of analytic services, their practice of setting PII to ASMs still jeopardizes users' privacy. Also, the top 3 market share holders have relatively less (31/120) misconfigurations maybe because they have instructions of ASMs in their documentation and TOS, which help avoid misconfigurations.

Misalignment of Apps' Privacy Policies

Misconfiguration of ASMs may also cause misalignment between an app's code and privacy policy. To find such apps, for each of the 120 apps that set PII to ASMs , three of the authors independently read the app's privacy policy and wrote arguments on why he / she believes using PII for analytic services is a potential policy misalignment or not. Then, the authors met to discuss the arguments for each app, and voted to determine whether the misconfiguration is misaligned with the privacy policy.

We found 27 out of 120 apps have misconfigurations that are misaligned with their own privacy polices. 15 apps vaguely mentioned in their privacy policy that they may share PII of users with third parties. 58 apps have no misalignment with their own privacy policies as they explicitly indicate that they will share specific personal information type to third-parties. The remaining 20 apps either have a non-English privacy policy or the privacy policy web-page is not available. The detailed discussion record of all 120 apps is available in our anonymous website, and misalignment examples are presented later in this subsection.

Privacy Misalignment. We consider an app to be misaligned with its privacy policy if the policy does not indicate that it will share PII with third parties, or if the policy claims anonymous data collection. For example, the social app Emojidom's privacy policy [61] states that:

> ***Do third parties see and/or have access to information obtained by the Application?***
> *Only aggregated, anonymized data is periodically transmitted to the analytics tools which help us evaluate the Application's usage patterns and improve the Application over time.*

However, our test results show that this app set user email addresses to Crashlytics, which is

misaligned with this privacy policy.

Vague Privacy Policies. Privacy policies should inform users about types of user information are shared with third parties. Third party analytic services also request app developers to make this sharing explicit in their apps' privacy policies. For example, Crashlytics is one of the most popular third party analytic services for helping developers to analyze crashes in their apps. Crashlytics requires that all developers maintain a privacy policy that fully and accurately discloses the type of information shared with Crashlytics [46]. Among 120 apps that send PII to analytic services, 15 of them abstractly indicate that they may share personal information to third-parties without specifying what the information types are. For example, the shopping app Staples sets user email address to Crashlytics and its privacy policy states that [63]:

> *We may share your Personal Information with our third-party service provider to process transactions or provide services on our behalf, including but not limited to providers of product delivery services (for example UPS and FedEx) and website analytics (for example Google Analytics).*

No misalignment We consider an app has no misalignment with its privacy policy if it clarifies the data types being shared with third-party service providers.

Finding 4. Our answer to **RQ4** is that, among 120 apps with misconfiguration of ASMs, the misconfigurations cause terms-of-service violation of analytic services in 31 apps, and privacy policy misalignment in 27 apps.

3.4.5 Threats to Validity

The major threat to internal validity of our study is the false positives and negatives in our misconfiguration detection process. Since we report only observed misconfigurations at run time, we should not have false positives. It is possible that our dynamic analysis failed to trigger some misconfigurations, our collected ASMs are not complete, or our matching process missed some

sophisticated transformed argument values. So our reported number of misconfigurations is actually an under-estimation, which will not undermine our major findings. To reduce this threat, we carefully scanned the documentation of analytic services, combined monkey and manual log-in to enhance the code coverage, and considered various value transformations when matching the reference user profile with system logs. Since most developers will perform configuration of analytic libraries when the app is started, the false negative rate caused by uncovered misconfiguration should not be high. The major threat to external validity of our study is that our findings may apply to only the 1,000 apps under study. To reduce this threat, we chose the top apps from Google Play store and these apps covers almost all different app categories.

3.5 Lessons Learned

In this section, we discuss the potential privacy risks found and our recommendations for different parties involved in the configuration of analytic services.

3.5.1 Privacy Risks

Although top analytic services advise app developers to not use PII as user attributes, many app developers still do so and no mechanism has been provided (either by Android or the analytic services themselves) to prevent app developers from using PII. This means that the analytic services may unintentionally link a behavior report to a specific individual. Based on our experiment results, a non-trivial number of apps are using emails and device identifiers (e.g., Android ID, IMEI, serials number) as user attributes. These identifiers are long-lived and can be used to construct a user's comprehensive profile from multiple apps using the same analytic service. Since most analytic services further share their collected data to third parties for business purposes, the personal-identifiable comprehensive profiles can be exposed to more risk due to the neglect of PII inside the data.

Since analytic services and app developers hold a large amount of valuable user data, it is very likely that they can be targeted for information theft/leakage attacks. When an information leakage

incident happens, if the data stored on the server is not in an anonymous and aggregated form, the consequence will be much more severe than the scenarios where they are anonymized and aggregated. Because analytic services do not expect app developers to set PII to the ASMs, they may not have corresponding mechanism to detect PII in the collected data, and thus may not use protection mechanisms (e.g., encryption) on the collected data.

3.5.2 Actionable Suggestions

Base on our study, the five parties involved in analytic services may take some counteractions to reduce the privacy risk caused by ASM misconfigurations.

Research Community In order to precisely and comprehensively detect misuse of analytic service ASMs, new static techniques are desired to detect the data flow from PII sources to the ASMs. Although it is possible to take advantage of off-the-shelf information flow analyses [79, 109], the challenge still remains of detecting PII sources and ASMs. For PII sources, many types of PII (e.g., username, user's email) are user defined so their source may be a text box, a local file or a database which cannot be easily differentiated from other non-PII information. Therefore, more precise techniques to identify PII sources or intermediate sources (e.g, a variable that loads PII values from a file or the database) are required. For ASMs, although we manually constructed ASM sets for 18 popular analytic services in the study, the analytic services are continuously evolving and new analytic services may become popular. For this reason, our sets can quickly become out-of-date. Therefore, novel techniques to automatically identify ASMs and their behaviors are desirable.

Another potential research endeavor is studying how analytic services can vet and anonymize PII so that they can enforce the privacy requirement of using ASMs. One challenge is that the analytic services do not know where the argument values come from. So, a likely solution is value-based detecting of PII, where a classification model may be learned to detect PII values in run-time arguments.

Privacy profiles [164] are automatically extracted from apps to provide fine-grained information of collected and shared information types, but they cannot handle advanced privacy properties

such as data anonymity and aggregation. Anonymity may be verified by checking whether data is combined (e.g., concatenated, put into one object or key-value pair) with PII. Aggregation may be verified by checking whether individual data is destroyed (e.g., freed) at the end after they are read.

App Developers. App developers should take more care in following ASM documentation/terms of service and avoid setting any PII as user attributes. Instead of using raw PII, developers could encrypt or hash the data before it is passed to analytic services or use non-PII instead. For example, if the differentiation of users helps on more precise statistics (e.g., how many users are using their app or certain activity), they can use Advertising IDs, randomly-generated IDs, or encrypted/hashed PII as user's identifier. App developers should pay attention to their privacy policies as well, as they need to make sure the policy is consistent with their practice of using analytic services. At the same time, a clear profile on what kind of PII is set to ASMs can help users understand how their privacy data can be used by analytic services.

Analytic Services. Analytic service developers should enhance and enforce data anonymity and aggregation in their code base. In particular, just like Google Analytics, Firebase, Flurry, and MixPanel, other analytic services should also try to provide a more clear and easily reachable instruction about privacy-aware configuration. Meanwhile, when designing and implementing methods, analytic service developers should avoid or limit the usage of over-broad/vague methods (e.g., `setProperties()`) and methods that are meant to receive PII (e.g., `setUserEmail()`, `setUserLocation()`). They should also add encryption features for methods that may receive PII from the app.

Second, when an app sets PII to ASMs, analytic services could have mechanisms to detect and anonymize the PII (e.g., regular expressions). In this way, analytic services could add vetting mechanisms in the implementation of ASMs to reject PII or raise warnings on detection. Alternatively, instead of transparently handing over the PII to app developers in their reports, they could encrypt the PII or replace it with other non-PII, and then perform analysis on the pre-processed data. After that, analytic services should generate a report that only contains aggregated data about user behaviors.

Platform Providers. The Android platform has applied some strategies to reduce the privacy risk over the years. For example, in Android version 8.0 and higher, Android ID is no longer a constant value for different apps installed on the same device. This mechanism helps to prevent the analytic services from gathering an individual user information across multiple apps. Since the Android platform has access to much PII for the device's users, such as Google email account, Android ID, Device ID, it should be able to vet such data sent to ASMs of analytic services. Working together with analytic services (e.g., asking them to annotate ASMs), the Android SDK could provide on-the-fly suggestions on which APIs and API options should be used while app developers are coding. Furthermore, the Android platform could provide the option to automatically reset the Advertising ID periodically for users.

App Users. App users should be aware that they can be un-anonymously tracked if app developers do not properly set their attributes on analytic service. Our study found that some app developers use usernames in analytic services so we suggest app users to avoid using their real names or PII when registering with different apps. In addition, Google Advertising ID has been encouraged to be used as the individual identifier in the analytic services. However, if a user does not reset the Advertising ID frequently, it becomes another long-lived online identifier. So we encourage app users to reset their Advertising IDs periodically to avoid being identified as the same individual for a long time period.

Chapter 4: UNCOVER DATA ACCESS AND REDUCE FALSE NEGATIVES OF STATIC TAINT ANALYSIS THROUGH HYBRID PROGRAM ANALYSIS

In the previous chapter, we discussed how we use our semi-automatic framework to observe sensitive data that apps share with certain types of third-party services. In order to automatically analyze data collection transmission in mobile apps, we investigated static taint analysis techniques, which traces information flow of tainted sensitive data access (source) to potential channels through which sensitive data could be leaked (sink). However, static taint analysis suffers from false negatives due to statically inaccessible code and incomplete source list. In this chapter, we proposed a novel approach to reduce such false negatives in static taint analysis. Specifically, we proposed a hybrid taint analysis approach to uncover the inaccessible sensitive data access and the leaks caused by them.

4.1 Overview

As a means to reduce false negatives in static taint analysis, we propose an approach that uses the results of dynamic taint analysis as additional sources to supplement static taint analysis. We implement and evaluate our approach for the Android platform because it has well-established static taint analysis tools [6, 7, 81] and downstream applications [126, 146, 201]. Although the effectiveness of such dynamic supplement is limited by the test coverage, our evaluation shows that it can reduce many false positives with a simple random testing strategy based on Monkey [65].

The base version of our approach is referred as DySTA (Dynamic Supplement of static Taint Analysis). DySTA first runs static taint analysis and dynamic taint analysis with the same set of initial sources, respectively. Once DySTA observes a variable holding a tainted value in the dynamic taint analysis that is *not* observed as tainted by the static taint analysis, the variable will be considered a new source (referred to as an *intermediate source* to be differentiated from the original sources). For the set of all intermediate sources, DySTA runs the static taint analysis again to find

additional taint flows. Unlike with static analysis, dynamic analysis is performed at run time, so it is less affected by blockers and is able to trace taint flows through dynamically loaded or generated code. Furthermore, even for pure black boxes (e.g., external flows through network servers or un-instrumented code), it is still possible to apply value-based dynamic taint analyses [132] which detect taint flows based on the observation of unique values preset at the source locations. As a result, DySTA retains the static taint analysis ability to trace all possible program paths outside of blockers which may not be triggered during testing while gaining the ability to detect traces *through* blockers thanks to the taint flows detected by dynamic taint analysis.

While the above approach can reduce false negatives, the basic design of DySTA has an important limitation. Since it simply concatenates static and dynamic taint flows without any constraints, the context sensitivity of the original static taint analysis will be lost. Therefore DySTA alone will lead to additional false positives besides those in the original static taint analysis for cases where blockers were analyzed. To overcome this, we further propose hybrid context matching in which the context of dynamic taint flows is injected into the intermediate sources. DySTA is then augmented so the subsequent static taint analysis considers only taint flows matching the injected context. By incorporating context matching, we implemented ConDySTA (Context-aware DySTA) as an extension of FLOWDROID, a state-of-the-art static taint analysis tool for Android apps. We evaluated DySTA and ConDySTA with REPRODROID, a benchmarking framework for Android analysis tools [158]. The results show that both DySTA and ConDySTA were able to reduce 12 out of the 28 common false negatives missed by all six static taint analyses considered in REPRODROID, and context preservation enabled ConDySTA to further eliminate all nine additional false positives reported by DySTA. We also performed a comparison of our approach and FLOWDROID on the 100 most downloaded Android apps according to PlayDrone [45]. Our evaluation showed that, with minimal testing and dynamic analysis, ConDySTA was able to detect 39 additional taint flows on top of the 281 taint flows reported by FLOWDROID. Furthermore, ConDySTA was able to preserve context sensitivity and rule out 1,029 taint flows with context mismatches from the detection results of DySTA.

This paper presents the following contributions.

- We demonstrate that dynamic taint analysis results can be used as a supplement to static taint analysis to reduce false negatives in practice.
- We developed a novel approach, ConDySTA, to preserve the context sensitivity of static taint analysis when supplemented by dynamic taint analysis.
- We performed evaluations using the REPRODROID benchmark and 100 top Android apps from Google Play demonstrating that ConDySTA can reduce many false negatives reported by state-of-the-art taint analysis tools and largely reduce false positives from our baseline solution.

4.2 Running Example

```
1  public String foo(String in){
2      blocker(in);
3      return foo2();
4  }
5  public String foo2(){
6      String inter = blocker2(); //an intermediate source
7      return inter;
8  }
9  public void bar(boolean flag){
10     String in = source();    //an original source
11     String out = foo(in);
12     if(flag){
13         sink(out);  //a potential taint flow
14         String in2 = "safe";
15         sink(foo(in2)); //a false positive
16     }
17 }
```

Listing 4.1: Static Taint Analysis False Negative Example

4.2.1 Running Example

Consider the example code in Listing 4.1. In the code, method `foo()` simply returns the value it receives as the argument. In particular, we assume that the parameter value of `foo()` is passed to `blocker(...)`, and the value is fetched in method `foo2()` by invoking `blocker2()`, and `foo2()` returns the fetched value, which is further returned by `foo()`. Here, we do not make assumptions about the implementation of `blocker(...)` and `blocker2()`, but one example of such an implementation can be the writing and reading files in the file system or tables in a

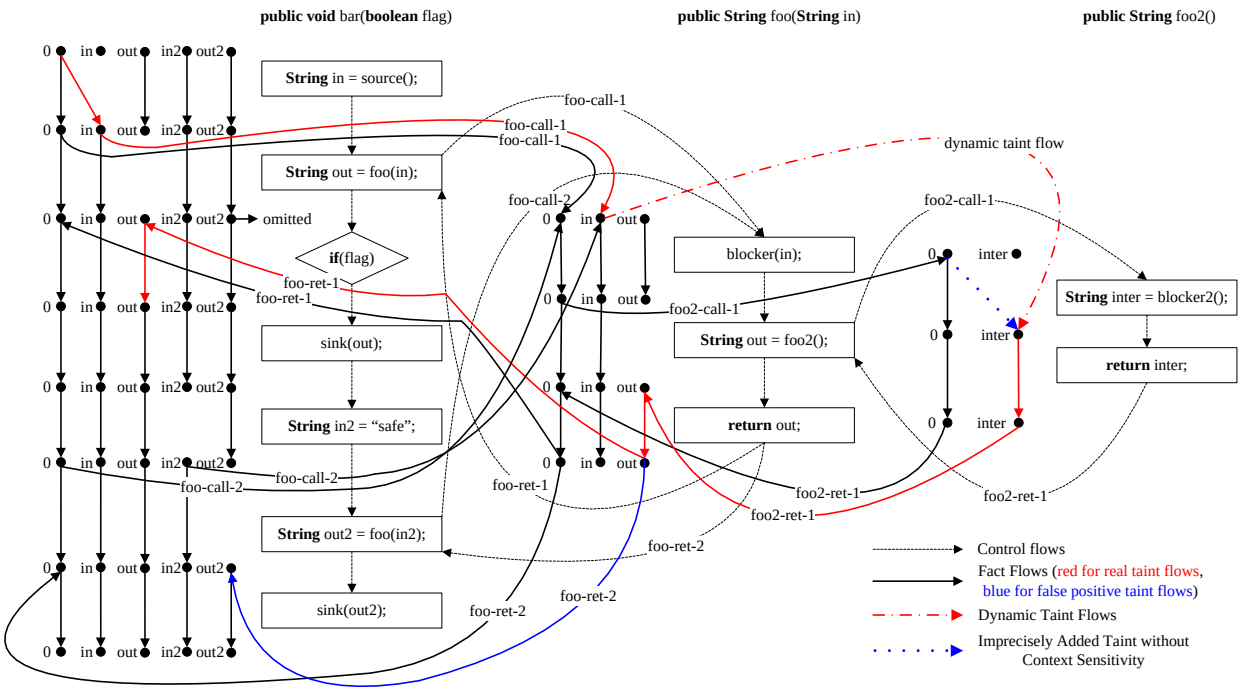


Figure 4.1: Analysis of the Running Example using IFDS Framework

database, respectively. Such a taint flow could not be traced as we assume blocker code portions (i.e., methods `blocker(...)` and `blocker2()`) are not accessible or analyzable by static taint analysis. Therefore, static taint analyses will not taint variable `inter` in Line 6 and will thus miss the taint flow from method invocation `source()` in Line 10 to `sink(out)` at Line 13.

4.2.2 DySTA Approach

Our basic solution, DySTA, executes the program and performs *dynamic* taint analysis after the initial *static* taint analysis. In the example, DySTA would taint variable `inter` at Line 6 as an intermediate source according to the result of the dynamic taint analysis which is able to follow the data flow through methods `blocker(...)` and `blocker2()`. Static analysis would then be applied again incorporating the intermediate source, thus detecting the taint flow at Line 13. However, since DySTA would not consider the calling context of `foo(...)` and `foo2()`, the taint would be further propagated to the expression `foo(in2)` at Line 15, although the argument `in2` passed in here is not a user information value from the original source method invocation

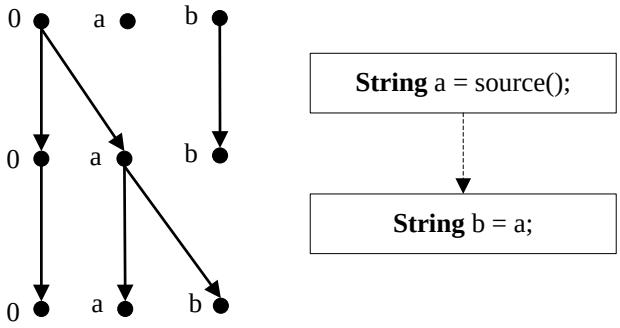


Figure 4.2: Illustration of Taint Flow Functions

at Line 10. Therefore, DySTA will detect an additional taint flow at Line 15. This false positive would be due to the second static taint analysis which incorporates the intermediate source from the dynamic analysis which does not include the calling context. So while the static taint analysis itself is context sensitive, the combination of dynamic taint analysis and static taint analysis becomes partially context-insensitive.

It should be noted that, because dynamic taint analysis cannot cover all possible paths, static taint analysis may be necessary to detect the taint flow at Line 13 (i.e., when parameter `flag` is not true during the execution). Furthermore, the lack of execution coverage on Lines 13-15 would make it impossible to rule out the false positive at Line 15 based on dynamic analysis alone (i.e., finding out `foo(in2)` at Line 15 is returning value "safe").

4.2.3 Code Analysis with the IFDS Framework

The IFDS framework, developed by Reps, Horwitz and Sagiv [169], defines a general mechanism to perform inter-procedural, flow-sensitive, and context-sensitive analysis. The framework is based on a program's inter-procedural control flow graph, referred as the "exploded super graph". The exploded super graph of our running example is presented in Figure 4.1. In Figure 4.1, we use dashed arrows to present control flows. Cross-procedure control flows are decorated with labels such as "foo-call-1", "foo-ret-1", and "foo-call-2" to differentiate call sites. For example, we can tell from the labels that call edge "foo-call-1" matches with return edge "foo-ret-1".

IFDS uses flow functions to represent transfer functions in flow-sensitive analysis on distribu-

tive finite properties. A flow function consists of a set of "from facts" and "destination facts", as well as arrows from the former to the latter. An arrow from fact a in the "from facts" to fact b in the "destination facts" indicates that if a holds before the statement is executed, b will hold after the statement is executed.

For example, Figure 4.2 shows flow functions of static taint analysis in which the facts are local variables (indicating that the variable is tainted or not), plus 0, a special fact that always holds. For statement `String a = source();`, the arrow from fact 0 to fact a indicates that variable `a` will be tainted no matter what (as 0 always holds). The arrow from fact b to fact b indicates that if b is tainted before the statement, then it is still tainted after its execution. Similarly, for statement `String b = a;`, the arrow from a to a indicates that whether a is tainted is unchanged before and after the statement, and the arrow from a to b indicates that if a is tainted before the statement execution, b will be tainted afterward. Given flow functions of all statements in the exploded super graph, the inference of a fact at a certain statement can be deduced to a graph reachability problem. In particular, it is a CFL reachability problem [168] because along the reachability path the arrows labeled with call-sites and return-sites must match to preserve context sensitivity.

In Figure 4.1, we show the flow functions of all statements in the three methods as solid arrows to the left of the control flow graph. Note that for method `bar(boolean)`, we omitted the fact for variable `flag` and the flow functions (and control flow) of the else branch to enhance the readability of the graph. From the figure, we marked as red the edges that form the taint flow from method invocation `source()` to the method invocation `sink(out)`. This flow cannot be detected by IFDS because it contains a dynamic taint flow path (presented as the red dash-dotted arrow on the top left) through `blocker(String)` and `blocker2()`, which cannot be statically analyzed at all. Without dynamic taint flow, IFDS finds no flows from the source to the sinks.

It should also be noted that if we simply add the dynamic taint flow path as an additional flow as shown in the graph, IFDS will still not identify the taint flow (marked in red), because the return edge "foo2-ret-1" will be mismatched with "foo-call-1" in this flow, and this flow is actually not along a feasible execution path as it directly goes from `foo(String)` to `foo2()`. An-

other possible solution is to add the whole dynamic execution paths inside `blocker(String)`, `blocker2()`, and their dependencies into the exploded super graph. However, since the code inside blockers are out of the box of the original static analysis, their transfer functions (i.e., flow functions) may be undefined. This can make the implementation of combined analysis very complicated and even infeasible. From figure 4.1, we can also see that, if we directly use `inter` as the source (i.e., adding the dotted blue arrow from fact 0 to fact $inter$), IFDS will identify the two flows to both `sink(out)` (true positive, marked in red) and `sink(out2)` (false positive, marked in blue), because IFDS allows unmatched call/return sites (feasible paths) but disallows mismatched call/return sites (infeasible paths).

4.2.4 Incorporating Context

In ConDySTA, we inject the dynamic calling context of an intermediate source s to the static taint analysis from s. In particular, the dynamic calling context of an intermediate source s consists of all the call-sites that have not returned on the dynamic taint propagation path from the original source to s. In the calling context, the call-sites are ordered in the same order as they are in the dynamic taint path. In our running example, the dynamic calling contexts of intermediate source `inter` at Line 6 will be `foo(in) at Line 11`, and `foo2()` at Line 3.

With the acquired dynamic calling context of s, in the following static taint analysis from s, ConDySTA will filter out the static taint propagation paths that do not match with the dynamic calling context. This is not a straightforward process due to recursive calls (for which there can be infinite static taint propagation paths). In particular, in the CFL-reachability [168] algorithm to solve the IFDS problem [169], besides finding feasible paths with matched call-site-return-site pairs (so that the paths are feasible with context sensitivity), we need to further identify the feasible paths containing a sequence of unmatched return-sites that match with the dynamic calling context C. We refer to such static taint-propagation paths as C-context-matching paths.

ConDySTA implements this by extending the exploded super graph in IFDS framework with a virtual flow to the intermediate source with the dynamic calling context as the edges. In the

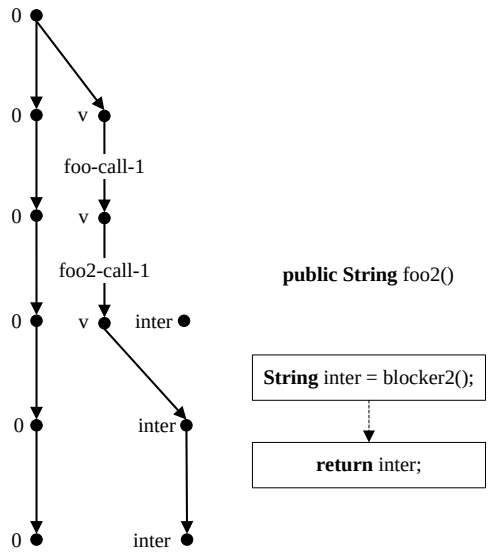


Figure 4.3: Illustration of ConDySTA Solution

extended graph, we can directly apply the standard CFL-reachability algorithm, and each taint propagation path from the original source in the extended graph can be mapped to a C-context-matching path. Figure 4.3 shows the solution of ConDySTA on the running example. From the figure we can see that we added a virtual fact v, that taints the intermediate source $inter$ and a number of virtual arrows on v before it taints $inter$. These virtual arrows are labeled with call-edges in the dynamic calling context, and are added in the reverse order so that they can match the return edges during the following IFDS analysis.

4.2.5 Lack of Dynamic Taint-Propagation Paths

Typical dynamic taint analyses propagate taints along with read/write accesses to memory locations along with program execution, so it is natural for them to record the dynamic taint-propagation paths. However, even this recording can sometimes be difficult in practice.

First of all, if the taint propagation is at the OS/hardware level [108, 154], it can be difficult to map the taint propagation paths back to source code due to multiple levels of abstractions. Even if a mapping is constructed, the mapping can be fragile and specific to a version of programming language runtime and OS. Second, the dynamic taint analysis itself may still miss some taint paths

through file systems, databases, and networks. Third and most importantly, unlike static taint analyses which are based on relatively stable programming language syntax/semantics, dynamic taint analyses need to work with most fine-grained system features and implementation details, so they can be easily out-of-date due to fast software evolution. For example, there have been two major dynamic taint analysis frameworks for the Android system: TAINTDROID [108] and TAINTART [187]. Neither of them support analysis on Android system versions above Android 6 (currently 8 and 9 are the most common Android versions [69]). Therefore, the simpler value-based dynamic taint analyses [132] often has better applicability. In particular, value-based dynamic taint analyses detect taint flows by inserting taints into the value fetched at the original sources (e.g., replacing the fetched value with strange values indicating the source location), or by changing data values at source locations in different executions and monitoring correlated value changes at other locations.

In all of these cases, ConDySTA may face a situation where the dynamic taint analysis can provide only tainted code locations, but not the taint propagation paths from the sources. So for the code in Listing 4.1, we can tell that variable `inter` at Line 7 is tainted, but we may not tell where the taint comes from and cannot extract the dynamic calling context from the dynamic taint-propagation path. To handle such cases, ConDySTA takes advantage of a key observation that *the dynamic calling context* of an intermediate source s is always a sub-sequence of the call stack trace of s. So we can directly extract the dynamic calling context from the stack trace, which is almost always accessible in dynamic taint analyses. For our example, the call stack trace for the intermediate source is as below.

```
at method foo2() (Line 7)
at method foo(String) (Line 3)
at method bar(boolean) (Line 11)
at some method (some line)
...
```

In the stack trace, the first three items actually provide the dynamic calling context: a call-site of `foo2()` at Line 3 of method `foo(String)` and a call-site of `foo(String)` at Line 11 of

method `bar(boolean)`. We can see that not all items of the stack trace belong to the dynamic calling context. For example, the call-site of `bar(boolean)` is not part of the dynamic calling context, because `source()` is invoked inside/after it, so the call edge for `bar(boolean)` does not need to be matched. On the other hand, if a call-site belongs to dynamic calling context, all call-sites above it in the stack trace are part of the dynamic calling context as the source value must go through these call-sites to reach the intermediate source as arguments, global variables, or value containers in blockers.

The key challenge is to decide how long a prefix of the stack trace needs to be in the dynamic calling context. The basic idea is that, if a call-site belongs to the dynamic calling context, it must be executed after the source location. Therefore, we can determine whether a call-site belongs to the dynamic calling context by checking the call stack of the source location or checking for tainted values in the reachable memory from the call-site.

4.3 Approach

In this section, we will first introduce the algorithm for DySTA and then present the construction algorithm for dynamic calling contexts in ConDySTA for propagation-based dynamic taint analysis. Finally, we will describe how dynamic calling contexts can be extracted for value-based dynamic taint analysis.

Before describing the approach, we provide the following static and dynamic taint analysis definitions. In our definitions, we use the term *expression location* to describe a pair of the form (*expr*, *line*) where *expr* is an expression and *line* is a description of where the expression is read or written in the code. For example, an expression location in our running example is (`inter`, Line 6).

Definition 1. ***Static Taint Analysis*** *We define a static taint analysis as a function* STA*: (*$Code$*,* $Srcs$*)* $\rightarrow$ $TaintLocs$*, where* $Code$ *is the code base to be analyzed and* $Srcs$ *are the set of expression location in* $Code$ *serving as the sources.* $TaintLocs$ *are a set of expression locations in* $Code$*.*

Definition 2. ***Propagation-Based Dynamic Taint Analysis*** *We define a propagation-based dynamic taint analysis as a function D_p: ($Code$, $Inputs$, $Srcs$) $\rightarrow$ $Paths$, where $Code$ and $Srcs$ are as defined in Definition 1, and $Inputs$ are input used to execute the code base. $Paths$ are a set of taint propagating program paths. Each path p in $Path$ is in the form of (s_1, s_2, ..., s_n), where $\exists$ $src \in Srcs$ such that s_1 reads src, and $\exists$ $i \in Input$ such that p is a contiguous subsequence of $exec(Code, i)$ (representing the execution path of $Code$ with input i), and the taint can be transitively propagated on p.*

Definition 3. ***Value-Based Dynamic Taint Analysis*** *We define a value-based dynamic taint analysis as a function D_v: ($Code$, $Inputs$, $Srcs$) $\rightarrow$ $LocStacks$, where $Code$, $Srcs$, and $Inputs$ are as defined in Definition 2. $LocsStacks$ are a set of pairs in the form of (loc, $stack$), where loc is an expression location that holds tainted value at least once in the execution, and $stack$ is a corresponding call stack when loc holds a tainted value.*

It should be noted that for both propagation-based and value-based dynamic taint analysis, one expression location may be tainted multiple times, and ConDySTA considers them as different intermediate sources if they have different taint propagating program paths or call stacks, because they may have different dynamic calling contexts which lead to different context matching in the following static taint analysis.

4.3.1 DySTA Algorithm

Based on the definitions above, our algorithm for DySTA is presented in Algorithm 1. The basic idea behind the algorithm is to first identify intermediate sources from the results of dynamic taint analysis (Lines 1-14), and then apply static taint analysis using them as sources (Lines 15-16). In particular, we first fetch the results of static taint analysis using original sources (Line 1), fetch the results of dynamic taint analysis (Line 2), and initialize the set of intermediate sources (Line 3). Then, for each statement in each taint-propagating execution path p (Lines 4-5), we first check whether the statement is re-entering statically analyzable code (Line 6). If so, DySTA checks which expression locations in that statement are tainted (Lines 7-8), and add those tainted

Algorithm 1: DySTA Algorithm

Require:
$Code$ is the code base to analyze
$Srcs$ is the set of source locations
$Inputs$ is the set of inputs for dynamic analysis
Ensure:
$TaintLocs$ is a set of tainted locations
1: $TaintLocs \leftarrow STA(Code, Srcs)$
2: $Paths \leftarrow D_p(Code, Srcs, Inputs)$
3: $interSrcs \leftarrow \emptyset$
4: **for all** $p \in Paths$ **do**
5: **for all** $s_i \in p$ **do**
6: **if** $\neg blocked(s_i) \wedge blocked(s_{i-1})$ **then**
7: **for all** expression locations $t \in s_i$ **do**
8: **if** $tainted(t) \wedge t \notin TaintLocs$ **then**
9: Add t to $interSrcs$
10: **end if**
11: **end for**
12: **end if**
13: **end for**
14: **end for**
15: $NewTaintLocs \leftarrow STA(Code, interSrcs)$
16: $TaintLocs \leftarrow TaintLocs \cup NewTaintLocs$

expression locations to the set of intermediate sources (Line 9).

DySTA extracts intermediate sources from only the statements re-entering statically analyzable code (referred to as *re-enter statements*) to avoid useless intermediate sources. In a statically analyzable segment of p, a taint on an earlier statement can be also statically propagated to tainted expression locations in later statements. Therefore, if static taint analysis using tainted expression locations in an earlier statement generates $result_e$, and static taint analysis using tainted expression locations in a later statement generates $result_l$, $result_e$ will be a strict super set of $result_l$. Thus, there is no need to extract intermediate sources from later statements. For similar reason, in Line 8, we do not consider as intermediate sources the expression locations that are already tainted by the original static taint analysis STA. In other words, we consider only the dynamic taint flows through blockers, which are not detectable by static taint analyses.

Algorithm 2: Construction Dynamic Calling Context

Require:
$path$ is a taint propagating program path
$InterSrcs$ is the set of intermediate sources
Ensure:
$ContextMap$ is a Hashmap from intermediate sources on $path$ to their corresponding dynamic calling context

```
1:  DContext ← ∅
2:  ContextMap ← ∅
3:  for all s_i ∈ p do
4:      for all expression locations t ∈ s_i do
5:          if t ∈ InterSrcs then
6:              ContextMap.Put(t, DContext.copy())
7:          end if
8:      end for
9:      if isCallSite(s_i) then
10:         DContext.push(s_i)
11:     else if isReturnSite(s_i) then
12:         DContext.pop()
13:     end if
14: end for
```

4.3.2 Dynamic Calling Context and Graph Extension

For ConDySTA, we extend DySTA with the matching of dynamic calling contexts. In particular, at Line 15 of DySTA algorithm, before calling STA to perform IFDS-based static taint analysis, ConDySTA inserts two processes. The first process extracts dynamic calling contexts for each intermediate source, and the second process extends the exploded super graph to add the dynamic calling context to it (see Section 4.2.4 and Figure 4.3). We present the algorithm we use to construct dynamic calling context from taint propagation paths as Algorithm 2.

The algorithm walks along the taint-propagating execution path (Lines 3-4), and collect all call-sites that have not returned in a stack $DContext$ (Lines 9-13). When an intermediate source t is reached (Line 5), ConDySTA copies the current $DContext$ and save it as t's dynamic calling context.

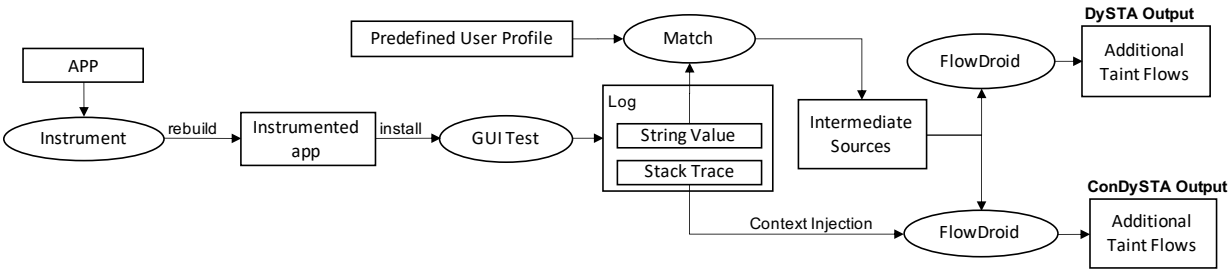


Figure 4.4: Implementation of ConDySTA

4.3.3 ConDySTA for Value-based Taint Analyses

When the taint-propagating execution path is not available (e.g., in value-based taint analysis), we cannot take advantage of the path to fetch the intermediate sources and the dynamic calling context. In such a case, we directly use the expression locations detected to hold tainted values as intermediate sources, and their call stack trace as dynamic calling contexts, as explained in Section 4.2.5. The challenge is to determine how many levels in the call stack trace (denoted as $stack_i$) belong to the dynamic calling context. Since only the open call-sites executed after the original source location need to be matched along the taint path, only items executed after the original source location need to be identified in the call stack trace.

If the source location is known, we can instrument the source location and fetch its call stack trace $stack_s$. Then we compare $stack_s$ and $stack_i$ to extract their common post-fix $post$. We can see that call-sites in $post$ are not-yet-returned call sites executed before the original source location, so $stack_i \setminus post$ will be the dynamic calling context to be matched. If the source location is not known, we cannot use the solution above. In this case, we can instrument all call sites in $stack_i$, and scan the reachable memory locations at the call site to check whether the tainted value can be observed. If the tainted value exists, we consider the call-site to be a part of the dynamic calling context, as the call-site should be executed after the source location is executed.

4.4 Implementation

In our implementation of ConDySTA, we use FlowDroid [78] for static taint analysis, as it is a state-of-the-art tool based on IFDS framework, and is compatible with the most updated Android system and apps. For dynamic taint analysis, we use value-based dynamic taint analysis, because the state-of-the-art propagation-based tools [108, 187, 215] are all out-of-date and do not work with Android 6.0 or higher (Android now is at 10). Although having the weakness of not handling control dependencies and encrypted data, value-based dynamic taint analysis also has its advantage on handling pure black boxes (e.g., web APIs whose implementations are on remote servers). Note that ConDySTA can always take advantage of new dynamic taint analysis once they are available. Figure 4.4 shows the implementation of DySTA and ConDySTA. They both first collect intermediate sources with dynamic analysis, and then detect additional taint flows using static taint analysis from intermediate sources. ConDySTA additionally checks whether an additional taint flow has a calling context matching with the dynamic calling context of the corresponding intermediate source.

4.4.1 User Profile For Tainted Values

Value-based dynamic taint analysis requires tainted values for sources. Specifically, we use the values in the user profile of an Android device as the tainted values. The information type and taint values are presented in Table 4.1.

4.4.2 Intermediate Source Collection

When collecting the intermediate sources, we instrument all return values of methods whose return types are `java.lang.String`. The reason is that all the tainted values are of string type and are stored in string variables. Although they are sometimes organized as fields in objects, there is often a method declared in the object's class to fetch the value of the sensitive data as a string. Due to performance concern, we only implemented the return value. In further research, we may apply static analysis or machine learning to select part of string-type parameters as instrumentation

Table 4.1: User Info and Corresponding Source

User Info	Source for FlowDroid
IMEI = "355458061189396"	android.telephony.TelephonyManager: java.lang.String getDeviceId()
Serial = "ZX1G22KHQK"	android.os.Build: java.lang.String getSerial () android.telephony.TelephonyManager: java.lang.String getSimSerialNumber()
AndroidID = "a54eccb914c21863"	android.provider.Settings.Secure: java.lang.String getString(android.content.ContentResolver,java.lang.String) android.provider.Settings.System: java.lang.String getString(android.content.ContentResolver,java.lang.String)
Email = "********@gmail.com"	android.accounts.AccountManager: android.accounts.Account[] getAccounts() android.accounts.AccountManager: android.accounts.Account[] getAccountsByType(java.lang.String)
PassWord = "******" UserName = "******"	android.os.UserManager: java.lang.String getUserName() android.widget.TextView: java.lang.CharSequence getText() android.widget.EditText: android.text.Editable getText() android.widget.TextView: android.text.Editable getEditableText()
language = "English"	java.util.Locale: java.lang.String getDisplayLanguage() java.util.Locale: java.lang.String getDisplayLanguage(java.util.Locale) java.util.Locale: java.lang.String getLanguage() java.util.Locale: java.util.Locale getDefault()
country = "US"	<java.util.Locale: java.lang.String getCountry() java.util.Locale: java.lang.String getDisplayCountry(java.util.Locale) java.util.Locale: java.lang.String getDisplayCountry() android.location.Address: java.lang.String getCountryName() java.util.Locale: java.util.Locale getDefault()
AdvertiserId = "fc1303d8-7fbb-44d8-8a68-a79ffac06fea"	com.google.android.gms.ads.identifier.AdvertisingIdClient.Info: java.lang.String getId ()
timezone_1 = "CST" timezone_2 = "Central Standard Time"	com.android.exchange.utility... getTimeZoneDateFromSystemTime(byte[],int) com.android.calendar.Utils: java.lang.String getTimeZone(android.content...) com.android.calendar.CalendarUtils$TimeZoneUtils:... getTimeZone(...) java.util.Calendar: java.util.TimeZone getTimeZone() java.util.TimeZone: java.util.TimeZone getTimeZone(java.lang.String) java.util.TimeZone: java.util.TimeZone getDefault() com.adobe.xmp.impl.XMPDateTimeImpl: java.util.TimeZone getTimeZone() android.util.TimeUtils: java.util.TimeZone getTimeZone(int,boolean,long...) java.text.DateFormat: java.util.TimeZone getTimeZone()
Manufacturer = "motorola"	android.os.Build.MANUFACTURER
NetWork = "Wi-Fi"	android.net.NetworkInfo: java.lang.String getTypeName()

points.

After instrumentation, we rebuild the smali code back into APK format for testing. We use the Android Debug Bridge (adb) to automatically install the rebuilt apps onto our test device, login with predefined profile if required, and use Monkey [65] to explore the app for 20 seconds. We use minimal testing in the implementation and evaluation of ConDySTA to check whether it can detect additional taint flows even with minimal testing. So our evaluation results actually show a lower estimation of the ability of ConDySTA, and equipping ConDySTA with more advanced testing may further enhance its effectiveness. During testing, we utilize the Android system log to record the return values and call stacks of String type methods. Table 5.3 shows an example where Line 1 shows the return value; Line 2 shows the method that be invoked (`com.facebook.internal.AttributionIdentifiers.`

Table 4.2: System log of String method

1	09-12 16:25:13.442 W System.err: java.lang.Exception: fc1303d8-7fbb-44d8-8a68-a79ffac06fea
2	09-12 16:25:13.443 W System.err: at com.facebook.internal.AttributionIdentifiers.getAndroidAdvertiserId (AttributionIdentifiers.java:1)
3	09-12 16:25:13.443 W System.err: at com.facebook.marketing.internal.RemoteConfigManager.run (RemoteConfigManager.java:5)
4	09-12 16:25:13.443 W System.err: at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1133)
5	09-12 16:25:13.443 W System.err: at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:607)
6	09-12 16:25:13.443 W System.err: at java.lang.Thread.run(Thread.java:761)

`getAndroidAdvertiserId`). The following lines show the call stack trace of this method. We consider a method as an intermediate source when its return value contains any user info in Table 4.1. In this example, the return value is the AdvertiserId, so we consider `getAndroidAdvertiserId()` as an intermediate source. We also check for concatenated, reversed and hashed format of the user info. For example, "355458061189396_ZX1G22KHQK" is a concatenation form of IMEI and Serial number.

Due to the essential weakness of value-based dynamic taint analysis, we will miss encrypted values. Please note that this can be resolved if ConDySTA is integrated with a propagation-based dynamic taint analysis tool (which is straightforward once such a tool is available). Furthermore, the taint flow of encrypted values are usually of less concern.

4.4.3 Applying FlowDroid

We run FLOWDROID with the original sources to detect statically tainted locations and rule them out intermediate sources, this reduces the source locations for the second round of static analysis and makes sure that ConDySTA always finds new taint flows. After we collected the intermediate sources, we feed these sources to FLOWDROID as new source locations, and run FLOWDROID again.

Context Match. Using the generated full path, we can perform context matching with the call stack trace. First, we convert the call stack trace to a call path. For the example call stack trace in Table 5.3, we generate the call path as Figure 4.5. Then, we will convert this call path to the form of exploded super graph as shown in Figure 4.3, and combine it with the exploded super graph generated by FLOWDROID before running it for static taint analysis.

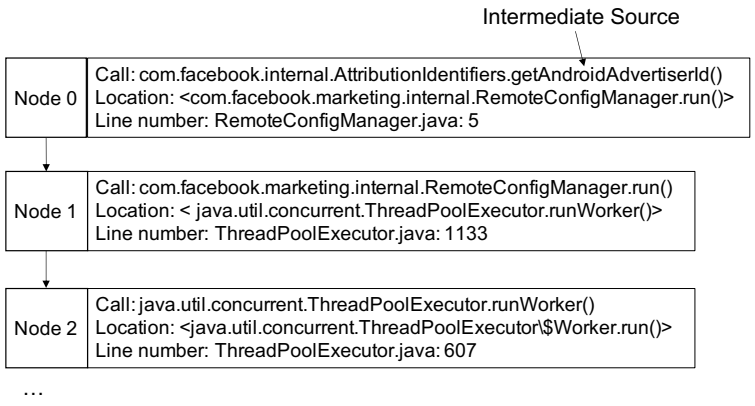


Figure 4.5: ConDySTA Call Path

4.5 Evaluation

In our evaluation, we consider two data sets. The first data set is a part of REPRODROID [157], a large and up-to-date benchmark which combines multiple earlier benchmarks [78, 90, 205] for static taint analysis for Android apps. REPRODROID's data set primarily consists of small apps with labeled taint flows (i.e., ground truth) written by researchers. The second data set consists of real-world apps from the Google Play store. We selected the 100 most downloaded apps that could be instrumented and successfully analyzed by FLOWDROID according to PlayDrone [45], a collection of meta data for Android apps on the Google Play store. The full list of the apps (as well as ConDySTA's implementation and detailed results) are available at our anonymized project website[1].

4.5.1 Research Questions and Summarized Answers

- **RQ1:** How many more taint flows can ConDySTA detect than static taint analysis alone? 28 taint flows across 28 apps from the benchmark were not detected by *any* of the six state-of-the-art static taint analysis . Of those 28 common false negatives, ConDySTA was able to detect 12. Among the 100 real-world apps, ConDySTA detected 39 more taint flows than FLOWDROID across 12 apps. FLOWDROID detected a total of 281 taint flows across 57

[1] `https://sites.google.com/view/condysta2020`

apps in total. The tainted information included email addresses, country, language, device's manufacturer, advertising ID, user's full name and username.

- **RQ2:** How many false positives can context sensitivity preservation reduce compared to naïve dynamic supplementation? In the benchmark evaluation, DySTA detected 21 taint flows (12 true positives and nine false positives) and ConDySTA reduced *all* of the nine false positive without missing any true positives. In the real-world app evaluation, ConDySTA was able to remove 1,029 taint flows with mismatched context from the result of DySTA.

- **RQ3:** Does ConDySTA detect taint flows not detected by the dynamic taint analysis itself? Among the 39 taint flows detected by ConDySTA, 19 of them were also directly detected (and thus confirmed) by dynamic taint analysis, the remaining 20 were additionally detected through static taint analysis.

- **RQ4:** How efficient is ConDySTA? The execution time of ConDySTA ranges from less than one second (when no intermediate sources are found) to 4,266 seconds, which is comparable to the execution time of FLOWDROID.

4.5.2 Evaluation on the Benchmark

A major challenge in evaluating program analysis on real-world applications is the lack of ground truth, so we first evaluate ConDySTA on the REPRODROID [158] benchmark, which consists of apps with taint flows labeled by earlier researchers. REPRODROID combines three existing benchmarks: DroidBench [80], ICCBench [205], and DIALDroidBench [90] and contains additional apps with code features not covered by the three benchmarks. The apps in REPRODROID are mostly written by earlier researchers and are simple enough for the researchers to manually identify and label all taint flows (i.e., pairs of source and sink locations[2]) in the app. The apps cover many different code features to check whether static analysis can handle handle those features. In the initial study [158] on REPRODROID, the authors evaluated six state-of-the-art static taint

[2]Multiple flows between the same pair of source and sink locations are considered as one.

analysis : AMANDROID [6], DIALDROID [7], DIDFAIL [3], DROIDSAFE [4], FLOWDROID [81], ICCTA [5], and reported detailed results.

Reducing False Negatives

In order to evaluate the effectiveness of ConDySTA on detecting additional taint flows, we applied ConDySTA on the apps (from REPRODROID) which contain at least one taint flow that cannot be detected by any of the six static taint analysis (i.e., a *common false negative* of all six). We consider these apps and taint flows because users can always combine existing static taint analysis and use the union of their results to reduce false negatives, and we want to check whether ConDySTA can further reduce false negatives on top of that. We identified 33 common false negative taint flows from 33 apps (one common false negative per app). Their covered code features include implicit taint flows, native code, reflection, and inter component communication. Among the 33 apps, five of them were out-of-date and thus could not be installed or crashed immediately upon execution (note that none of the labeled taint flows in these apps are observed), so they were excluded. We applied ConDySTA on the remaining 28 apps, and it detected 12 correct taint flows and thus reduced 12 out of the 28 common false negatives. We also applied DySTA on the 28 apps and it detected the exact same taint flows, thus showing that by adding context sensitivity, ConDySTA did not introduce false negatives.

Table 4.3 shows the details of these 12 taint flows. In the table, the four columns represent their IDs in the benchmark, covered code features, enclosing apk names, and source/sink pairs. In flow 124 (ImplicitFlow), the data has been converted into an array of `Char` and copied back to another string before flowing to the sink. In flows 191 and 192 (Native code), native code is used to fulfil part of the taint flow. In flows 203 and 206-209, part of the taint flow (sending information through intents) is fulfilled with method invocations performed by reflection along with dynamic generation of class/method signatures used in reflection. In flows 24, 25, 27 and 32, the data is transmitted across components through intents. Notably, the latter two code features (Reflection and ICC) are supported by some of the tested (e.g., AMANDROID [6],

DIALDROID [7], and ICCTA [5]), but they are only partially supported so some complicated cases cannot be handled as shown in the result of ReproDroid study [158]. ConDySTA detected these 12 flows based on intermediates sources such as `de.ecspride.ImplicitFlow1: java.lang.String copyIMEI(java.lang.String)`, `android.content.Intent: java.lang.String getStringExtra(java.lang.String)`, etc. From the result, we can also see that ConDySTA is independent from code features (i.e., types of blockers), so it reduces common false negatives caused by various types of blockers.

ConDySTA failed to reduce the remaining 16 common false negatives, simply because the corresponding taint flows do not involve any string type return values (which are the only instrumentation points of ConDySTA) and thus ConDySTA fails to detect intermediate sources. If ConDySTA instrumented all string type parameters, it would be able to detect all 28 of the common false negatives. However, we did not implement ConDySTA to instrument all string parameters due to the high overhead (due to the need to extract call-stacks at all instrumentation points) in real-world apps, resulting in a lack of scalability. In real-world apps, taint flows are much longer and more complicated so a string type return value is more likely to be involved. Upon further research, we may select only part of string-type parameters and other-type variables as instrumentation points, which may realize the full potential of ConDySTA.

False Positives

To evaluate ConDySTA's performance on reducing false positive caused by context insensitivity, we applied both DySTA and ConDySTA on 43 apps from REPRODROID that contained at least one true negative. When constructing REPRODROID and earlier benchmarks, researchers also labeled fake taint flows (i.e., pairs of sources and sinks without a flow between them). These labeled true negatives can be used to check whether static analysis report false positives. Of the 186 such fake taint flows in the 43 apps, DySTA mistakenly reported nine of them. With context matching, ConDySTA did not report any of them, so it reduced nine false positives of DySTA to zero. Note that ConDySTA is implemented to supplement a static taint analysis (i.e., FlowDroid),

Table 4.3: False negative taint flows detected by ConDySTA

ID	Feature	Apk	Source & Sink
DroidBenchExtend			
124	ImplicitFlows	ImplicitFlow1	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.i(java.lang.String,java.lang.String)
191	Native	SinkInNativeLibCode	android.telephony.TelephonyManager.getDeviceId()
			mod.ndk.ActMain.cFuncSendData(java.lang.String)
192	Native	SourceInNativeCode	mod.ndk.ActMain.cFuncGetIMEI(android.content.Context)
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
203	Reflection_ICC	OnlyIntent	android.telephony.TelephonyManager.getDeviceId()
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
206	Reflection_ICC	OnlyTelephony	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
207	Reflection_ICC	OnlyTelephony_Dynamic	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
208	Reflection_ICC	OnlyTelephony_Reverse	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
			android.telephony.SmsManager. sendTextMessage(java.lang.String, ...)
209	Reflection_ICC	OnlyTelephony_Substring	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
ICCBench			
24	IccTargetFinding	icc_dynregister1	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.d(java.lang.String,java.lang.String)
25	IccTargetFinding	icc_dynregister2	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.d(java.lang.String,java.lang.String)
27	IccTargetFinding	icc_explicit1	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.d(java.lang.String,java.lang.String)
32	IccTargetFinding	icc_implicit_mix1	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.d(java.lang.String,java.lang.String)

and the static taint analysis itself may report false positives which are not caused by ConDySTA.

4.5.3 Evaluation on Real World Apps

Five of the six used in the REPRODROID benchmark could not be applied to our real-world app dataset as four of them (AMANDROID, DIDFAIL, DROIDSAFE, and ICCTA) do not execute on recent apps[3] [158], and one (DIALDROID) targets only inter-app taint flows and not general intra-app taint flows. For these reasons, we were able to benchmark ConDySTA against those apps using REPRODROID's compatible test set, but we could *not* use REPRODROID to compare ConDySTA's performance against these for modern apps. To address this and present a more complete evaluation, we also evaluated ConDySTA on current real-world apps and compared it with the remaining working ConDySTA, FLOWDROID, to test ConDySTA's relevance to the current app landscape.

For fair comparison, we count taint flows the same way as FLOWDROID. In particular, multiple

[3]They support up to Android API level 19, and Android API is currently at level 29

Table 4.4: Taint flows detected by ConDySTA in real-world apps

App Package Name	Size (KLOC)	FlowDroid	DySTA	ConDySTA (Dynamic)	ExecTime(s) DySTA+ ConDySTA	ExecTime(s) FlowDroid
com.amazon.mShop.android.shopping	10881	1	25	2(0)	25	257
com.dianxinos.dxbs	3034	1	77	15(6)	4266	1162
com.disney.WMWLite	1489	2	11	2(2)	357	131
com.forthblue.pool	1778	3	22	2(0)	1630	270
com.gameloft.android.ANMP.GloftDMHM	2540	20	3	3(0)	29	18
com.mxtech.videoplayer.ad	4044	3	4	1(1)	574	27
com.pinterest	5534	0	2	4(4)	95	138
com.sgiggle.production	6015	0	1	1(0)	44	32
com.tubitv	7660	0	5	3(2)	38	273
com.waze	2996	1	1	1(0)	16	115
org.mozilla.firefox	2155	24	74	4(4)	18	1265
paint.by.number.pixel.art.coloring.drawing.puzzle	4795	0	14	1(0)	23	64
...	...	...	...		...	
Total	N/A	281	1068	39(19)	N/A	

taint flows between the same pair of source and sink locations are counted as one taint flow. So even if ConDySTA detects a different taint flow for a pair of source and sink locations between which FLOWDROID already detects a flow, we do not consider ConDySTA to have found a new taint flow. Furthermore, we use the configuration of FLOWDROID with context sensitivity and least false negatives (FLOWDROID has some configurations sacrificing soundness for performance). Finally, we make sure ConDySTA and FLOWDROID use the same set of sources and sinks. Column 2 of Table 4.1 shows the sources we use for each user information type. Note that full name, user name and password are provide through user input, so we use the `EditText.getText()` method invocations of the corresponding UI widget as the sources. To further confirm, we instrumented the `EditText.getText()` method invocations and print out the value passed in to make sure our input values are caught by these sources.

We present our evaluation results on additionally-detected taint flows in Table 4.4. In the table, Columns 1-5 present the name of the app, the size of the app in thousands of lines of smali code (note that we have only the byte code of apps as they are closed source), the number of taint flows detected by FLOWDROID, the additional number of taint flows detected by DySTA, the additional number of taint flows detected by ConDySTA (with the number of taint flows that also detected by dynamic taint analysis within these flows in brackets), and the execution time. Note that as we have 100 apps and limited space, we present only the apps with at least one taint flow detected by

ConDySTA. The full results are available in Tables 4.5 and 4.6. Also, for the execution time, we include only the following context-aware static taint analysis portion. Since the execution time of dynamic taint analysis largely depends on the testing intensity (and we are using minimal testing in our evaluation), it does not make much sense to combine the execution time.

Additionally Detected Flows Over FlowDroid

Among the 100 apps tested, FLOWDROID detected 281 taint flows using the Android platform sources, while ConDySTA detected 39 more taint flows. 19 of these 39 were confirmed with dynamic taint analysis and eight of the remaining can be manually confirmed (see Section 4.5.3 for more detailed inspection results). From Table 4.4, we can see that these 39 flows are distributed over 12 different apps. This shows that the practical complexity that causes unsoundness of static taint analysis is very common among top Android apps. For some of the apps (e.g., com.dianxinos.dxbs), FLOWDROID detects zero or very few taint flows while ConDySTA detected many, which shows that ConDySTA may be very helpful for some apps where blockers are used intensively.

ConDySTA vs. DySTA

A comparison between Columns 4 and 5 in Table 4.4 shows the benefit of ConDySTA. In particular, ConDySTA reduced 1,029 context-mismatched taint flows from 49 apps. So we can see that the reduction of context-mismatched taint flows happens in almost all of the apps. It should be noted that a context mismatched taint flow may not necessarily be fake. In very rare cases, the taint flow may happen under a different context not covered by dynamic taint analysis or even through another intermediate source not observed in dynamic taint analysis. However, we believe they should be removed because they should not be inferred from observed facts of the dynamic taint analysis. As an analogy, a weather forecaster may have a flaw so that Wednesday's weather is always reported to be stormy, which could be true in rare cases, but the flaw and corresponding forecasting results should be removed because they are not results from the forecasting model

Table 4.5: Taint flows detected by ConDySTA in real-world apps

App Package Name	Size (KLOC)	FlowDroid	DySTA	ConDySTA (Dynamic)	ExecTime(s) DySTA+ ConDySTA	ExecTime(s) FlowDroid
art.coloringpages.paint.number.zodiac.free	4348		11		1402	16
com.abtnprojects.ambatana	6094	2	4		620	75
com.adobe.reader	2084	3	45		4841	114
com.amazon.mShop.android.shopping	10881	1	25	2(0)	25	257
com.appsci.sleep	4815	7	5		65	807
com.arlo.app	7178	1	33		1946	197
com.audible.application	7531				2319	2383
com.audiomack	6796	1			18	19
com.aviary.android.feather	2579	5	31		1761	134
com.bbm	8208	1			20	2318
com.bfs.papertoss	2089	7	7		125	32
com.bydeluxe.d3.android.program.starz	5022		6		3319	10
com.calm.android	6352	1	1		209	487
com.cbs.app	8355				19	16
com.chewy.android	2873				23	20
com.classdojo.android	6088	1			19	120
com.cleanmaster.mguard	8771	15	11		2993	536
com.clearchannel.iheartradio.controller	8188				24	25
com.contextlogic.wish	2943	2	19		6082	1259
com.creativemobile.DragRacing	5630	3	4		4069	310
com.creditkarma.mobile	4594		8		14	47
com.devuni.flashlight	2371	1	11		1762	28
com.dianxinos.dxbs	3034	1	77	15(6)	4266	1162
com.discord	3238		8		922	105
com.disney.WMWLite	1489	2	11	2(2)	357	131
com.domobile.applock	2393	2	7		1268	31
com.dropbox.android	5656				56	36
com.drweb	2393	1			1868	26
com.duolingo	4309	1	15		347	191
com.ebay.mobile	8050				21	27
com.enflick.android.TextNow	9949		1		20	151
com.espn.score$_center$	966				26	203
com.facebook.mlite	2326	5			20	361
com.fingersoft.hillclimb	4468		11		36	21
com.forthblue.pool	1778	3	22	2(0)	1630	270
com.fox.now	5085		7		25	39
Total	N/A	281	1068	39(19)	N/A	N/A

Table 4.6: Taint flows detected by ConDySTA in real-world apps, Cont.

App Package Name	Size (KLOC)	FlowDroid	DySTA	ConDySTA (Dynamic)	ExecTime(s) DySTA+ ConDySTA	ExecTime(s) FlowDroid
com.game.JewelsStar	2946	2	6		45	12
com.game.SkaterBoy	2571	2	14		80	14
com.gameloft.android.ANMP.GloftDMHM	2540	20	3	3(0)	29	18
com.gameloft.android.ANMP.GloftIAHM	1596	49	40		2158	19
com.gau.go.launcherex	6996	8	41		919	140
com.gau.go.launcherex.gowidget.weatherwidget	4065	8	41		564	171
com.gonoodle.gonoodle	2736	1			17	18
com.goodrx	3883		3		76	160
com.gotv.nflgamecenter.us.lite	6308	2			17	324
com.groupme.android	2942	7	3		77	394
com.grubhub.android	4995		18		26	404
com.hulu.plus	5101				22	304
com.ibotta.android	8898				1521	14
com.imangi.templerun	2425				27	16
com.imangi.templerun2	2403		3		36	13
com.indeed.android.jobsearch	2066				33	39
com.kakao.story	3471	1	8		2150	237
com.konylabs.capitalone	5125	3			236	152
com.life360.android.safetymapd	5844	2	1		51	53
com.mcdonalds.app	8329				21	39
com.microsoft.appmanager	12102	2			26	250
com.microsoft.office.outlook	8169				74	90
com.mxtech.videoplayer.ad	4044	3	4	1(1)	574	27
com.naver.linewebtoon	6744		11		28	37
com.netflix.mediaclient	4682	3			35	7
com.offerup	8054				43	128
com.outfit7.talkinggingerfree	7787				25	14
com.outfit7.talkingtom	7990				30	20
com.outfit7.talkingtom2free	7958				36	26
com.pandora.android	13149				28	19
com.particlenews.newsbreak	3787	4	16		1016	222
com.picsart.studio	10604	1			41	41
com.pinterest	5534		2	4(4)	95	138
com.pof.android	3533	1	2		34	22
com.popshow.yolo	2801	1			16	67
com.poshmark.app	5163		4		16	13
com.postmates.android	2947		87		1203	515
com.roidapp.photogrid	6867	1	25		3340	750
com.roku.remote	4133				18	82
com.rovio.angrybirdsseasons	1814	1	7		33	23
com.sgiggle.production	6015		1	1(0)	44	32
com.shootbubble.bubbledexlue	1460	6	33		1609	187
com.skype.raider	2563	2			19	15
com.squareup.cash	3654		5		28	293
com.supercell.clashofclans	1141		3		14	3
com.supercell.hayday	1323		3		14	4
com.surpax.ledflashlight.panel	3101	1	31		1165	31
com.tencent.mm	13678	1	1		42	49
com.topfreegames.bikeracefreeworld	4423				70	21
com.tubitv	7660		5	3(2)	38	273
com.UCMobile.intl	6924	7			31	46
com.venmo	4018	3	111		9820	1064
com.viber.voip	2741	12	10		3876	1813
com.waze	2996	1	1	1(0)	16	115
com.yahoo.mobile.client.android.mail	5851				13	441
com.zillow.android.zillowmap	5331				15	5
flipboard.app	3406				569	291
jp.naver.line.android	13113	19	18		126	77
me.pou.app	1923	7			22	159
org.mozilla.firefox	2155	24	74	4(4)	18	1265
paint.by.number.pixel.art.coloring.drawing.puzzle	4795		14	1(0)	23	64
scratch.lucky.money.free.real.big.win	4571	1	30		1220	349
us.ozteam.bigfoot	4628		10		22	426
vStudio.Android.Camera360	6692	9			43	1920
Total	N/A	281	1068	39(19)	N/A	N/A

(which may be imperfect by itself). Note that in our evaluation on REPRODROID, all the removed context-mismatched taint flows are fake flows.

Comparison with Pure Dynamic Taint Analysis

We further studied whether ConDySTA detects only the taint flows that are already detected by dynamic taint analysis. If so, its value would be diminished. Among the 39 taint flows detected by ConDySTA, we instrumented the sink methods and applied dynamic taint analysis to check how many taint flows could be detected. The results are presented in the brackets of Column 5 in Table 4.4, which shows that 19 taint flows can be detected (and thus confirmed as true positives) and the remaining 20 cannot be detected. This shows that ConDySTA does provide more value by performing static taint analysis from the intermediate sources.

Execution Time

Finally, we recorded the execution time of ConDySTA (see Column 6 of Table 4.4). We can see that the execution time is within 5,000s, and for most of the apps it ranges from several hundred seconds to thousands of seconds. This is similar to those of FLOWDROID. Notably, as ConDySTA invokes FLOWDROID for intermediate sources, the largest portion of its execution can be attributed to FLOWDROID. It should be noted that DySTA+ConDySTA sometimes take much longer time than simply running FLOWDROID because of the additional intermediate sources.

Qualitative Analysis

To understand why FLOWDROID has the false negatives that ConDySTA detected, we further performed a qualitative analysis on the taint flows detected by ConDySTA but not FLOWDROID. Among the 39 taint flows, 23 flows are in apps which are heavily obfuscated and we were not able to understand the full taint paths (Note that 11 of the 23 flows were confirmed in dynamic taint analysis). Among the remaining 16 flows that we managed to fully understand, six flows were missed by FLOWDROID because the data flowed through the network (sent to remote servers

and fetched back), four flows were not detected because the data flowed to local cache files and were later read back, and six flows were not detected due to FLOWDROID's flawed modeling of `HashMap.putAll()`, which we confirmed with a trivial app with only this function on the taint path. Note that `HashMap` is particularly difficult to handle in static analysis as it can easily create many false positives if the entire `HashMap` is conservatively tainted. Finally, we can see that the blockers in real-world apps are very different from those pre-defined in REPRODROID. So ConDySTA's independence of blocker types can be an important benefit when applied to real-world apps.

4.5.4 Threats to Validity

One major threat to the internal validity comes from value-based taint analysis. Due to coincident string matches, some of the detected false negatives may not be real false negatives. To reduce such threat, we use complicated profile data to avoid coincident matches, and manually confirmed all detected false negatives on ReproDroid, and a large portion of those from real-world Android apps. One major threat to the external validity comes from the size and variety of our subject apps. To reduce such threat, we consider both a large existing benchmark and top real-world Android apps.

4.6 Discussion

Generality on Dynamic Taint Analysis. Since ConDySTA needs only intermediate sources (nodes on the taint paths) and their calling contexts (method invocations along the taint paths) from the dynamic taint analysis, ConDySTA should be able to directly take the output of any propagation-based dynamic taint analysis. Even if the method invocations along the taint paths are not provided by the dynamic taint analysis tool (which is unlikely for propagation-based analysis), ConDySTA can still directly use the system stack traces at intermediate sources as estimated calling contexts (just as how it handles value-based dynamic taint analysis). So, once a new dynamic taint analysis framework becomes available, ConDySTA can easily take advantage of it.

Generality on Static Taint Analysis. ConDySTA uses FlowDroid as the static analysis tool to be supplemented because it is context-sensitive, very robust to be still able to handle most Android apps on the market, and has been adopted by many downstream research efforts (e.g., [126] and [201]). DySTA integrates with static taint analysis by providing intermediate sources as new sources, so it can be directly used with almost any static taint analysis tools (as long as they allow adding new sources) without any effort. ConDySTA further encodes calling context into the inter-procedure control-flow graph in the IFDS framework, so it can be directly integrated to any IFDS-based static taint analysis. ConDySTA can be further adapted to integrate with more broader categories of static taint analyses by encoding the calling context into the intermediate code representation the analyses are based on.

Chapter 5: UNCOVER DATA ACCESS AND REDUCE FALSE NEGATIVES OF STATIC TAINT ANALYSIS THROUGH MACHINE LEARNING TECHNIQUES

In the previous chapter, we presented the design and evaluation of our framework ConDySTA, which use hybrid taint taint analysis to discover sensitive data access and detect leaks. Since ConDySTA rely on dynamic testing to trigger sensitive data related event, it is limited on the test coverage, one data access can be missed if the method was not triggered during testing. Also, ConDySTA requires human effort to register and login to the app, making it not feasible for use on a large scale. To avoid human effort and the limitation on test coverage, in this chapter we discuss about our framework, DAISY, which uses machine learning techniques to identify sensitive data access through methods defined by app and third-party libraries, specifically, to identify app's or third-party libraries' methods that return sensitive information.

5.1 Overview

Taint analysis has been widely used to identify privacy leaks. Many approaches [161,183] focus on platform information extracted from the Android device through the Android API. Because the Android API is relatively stable, these approaches depend upon existing lists of Android API methods that return sensitive data (e.g., `Android.telephony.TelephonyManager.getDeviceID()` returns the phone's IMEI), which is provided to static or dynamic analysis tools [79, 109] to determine the destination of sensitive data flows. By analyzing these flows, it is possible to determine what sensitive data is potentially accessed and collected from the app. More recent work [126, 151, 201] further traces the flow of user input data, which cannot be identified using the Android API alone, and requires tracing potentially sensitive data through GUI API method executions (e.g., `android.widget.EditText.getText()`). This tracing requires classifying the information types by first analyzing the GUI hierarchy [171] and then classifying labels associated with the method invocations.

These approaches have limitations because the Android API and user input of the app do not cover all possible sources of sensitive data. For example, an app may acquire data from back-end servers (e.g., from user profiles that were previously collected through a web portal, a different app from the same organization, or even a hand-written registration form) and then share this data with other entities through the app. As another example, an app may also fetch data from third-party services or apps (e.g., Facebook profiles) or use third-party services to store / process data (e.g., Firebase Storage) and fetch the stored / processed data back later. All of the sources in the above examples are not Android API methods, but methods from either third-party libraries or the app itself. Therefore, predefined lists of sensitive data sources in the Android platform can be insufficient, and an approach to identify additional sources beyond Android platform / GUI API methods is desirable.

To more thoroughly trace sensitive data in Android apps and supplement existing approaches, we propose **DAISY** (Dynamic-Analysis-Induced Source DiscoverY for sensitive data), which is a novel approach to automatically identify methods in the apps or third party libraries that return sensitive data (or a sensitive method). Below, we discuss the three major technical challenges to overcome in the application of natural language processing and machine learning for code analysis in DAISY along with the intuition behind our solutions.

- **Constructing a sufficiently large training set.** While automatic extraction of methods from many Android apps is easy, manually labelling each method as sensitive / non-sensitive could be prohibitively expensive due to the complexity of code semantics and sparsity of sensitive method (hundreds of non-sensitive methods may need to be reviewed before one sensitive method is found).

  Solution: DAISY overcomes this challenge by dynamic-analysis-induced automatic labelling. During training, we run all training apps and collects run-time return values of methods being executed. Then all the executed methods can be automatically labeled by testing whether their return values contain planted sensitive data (we can plant sensitive data such as device ID and account email address before running the apps).

Table 5.1: Example of context-method

Return Value: xxxxxxxxxx@gmail.com
Call Stack:
com.tubitv.helpers.PreferenceHelper.getString(PreferenceHelper.java:2)
com.tubitv.helpers.PreferenceHelper.getString(PreferenceHelper.java:3)
com.tubitv.helpers.UserAuthHelper.getEmail(UserAuthHelper.java:1)
...

- **Handling partially sensitive methods.** Some methods can be partially sensitive because they sometimes return sensitive values but do so only under certain conditions. For example, in the call stack in TABLE 5.1, it is impossible to determine the information type of the `PreferenceHelper.getString(o` method at the top because it is used across multiple contexts returning either sensitive or non-sensitive data. To address this issue, we consider a method's calling context (`UserAuthHelpr.getEmail(...)`), to infer that `PreferenceHelper.getString(...)` may return email address.

  Solution: Instead of just classifying single method, DAISY classifies methods along with calling contexts (called *in-context methods*). The same method with different calling context can be labelled differently during training and predicted differently during testing.

- **Recognizing text semantics in method signatures.** We consider methods' signatures as natural language sentences to leverage the advances in natural language processing (NLP) and machine learning to classify a method. While a word embedding provides a robust semantic representation of text, it can hardly handle words not seen in the training set, which are common in method signatures with informal abbreviated texts.

  Solution: We handle informal texts by taking advantage of the sub-word embedding feature for FASTTEXT [89, 130] framework. Sub-word embedding further considers sub-strings of words when constructing the word embedding so that an unseen word with a seen sub-string can still be properly represented.

We trained DAISY with 200 top-ranked apps (all rankings are based on PlayDrone [197]). The learned models are tuned by searching the hyper-parameter space of the training process for optimized hyper-parameter values that leads to best performance on the validation set. After this,

we applied DAISY on 2,816,751 in-context methods statically extracted from the call-graphs of 100 apps ranked from 201 to 300, and discovered 26,927 potentially sensitive in-context methods. Since it is virtually impossible to manually label all discovered sensitive in-context methods, we chose two subsets. The first subset (high-confidence set) consists of 170 in-context methods which are predicted by DAISY with the highest confidence for different considered context lengths and information types (up to top 10 for each combination). This subset evaluates the effectiveness of DAISY when a user is interested in only the most-likely sensitive methods (e.g., when an user has limited time or resources for scanning a batch of apps). The second set (random set) consists of 144 in-context methods which are randomly sampled (with 20% rate) from in-context methods of the top 5 apps (i.e., apps ranked 201 to 205) in our testing app set. This subset evaluates the effective of DAISY when a user is interested in all sensitive methods (e.g., when an app developer seeks to avoid privacy violations). The evaluation results show that DAISY is able to achieve an average precision of 75.3% for the high-confidence set and an average precision of 41.0% for the random set. Further analyses of the confirmed new sources show that (1) among 187 detected and confirmed new sources, 178 can be detected by neither value-based dynamic analysis nor static taint analysis and (2) further considering calling contexts of length 2 and 3 helps to discover 24 and 12 more new sources, respectively.

The contributions in paper are summarized as follows:

- A novel approach, DAISY, to discover sensitive methods in Android apps along with their calling context based on machine learning and sub-word embedding.
- An automatic labeling technique based on dynamic exploration of app code to extract large-scale training data sets from real-world apps.
- Viability for Android app marketplaces and developers to discover sensitive sources defined in the third party libraries and apps. Our evaluation shows DAISY yielding a significant number of manually confirmed new sources that can be used in static and dynamic taint analysis.

```
--------- beginning of crash
FATAL EXCEPTION: main
Process: com.zenga.zengatv, PID: 13218
android.view.WindowManager$BadTokenException: Unable to add window -- token
android.os.BinderProxy@189f3cf is not valid; is your activity running?
        at android.view.ViewRootImpl.setView(ViewRootImpl.java:679)
        at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:342)
        at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:93)
        at android.widget.Toast$TN.handleShow(Toast.java:459)
        at android.widget.Toast$TN$2.handleMessage(Toast.java:342)
        at android.os.Handler.dispatchMessage(Handler.java:102)
        at android.os.Looper.loop(Looper.java:154)
```

Figure 5.1: A sample call stack.

- Multiple manually and automatically labeled data sets of in-context methods with sensitive information types that can be leveraged in future research.

5.2 Call Stack and Calling Context

5.2.1 Dynamic Call Stack

A call stack refers to the actual method invocation sequence that happens during runtime. Developers can print a call stack at any point in the app code using provided API methods. For example, a call stack can be generated when the app crashes due to an error or exception, providing a list of method calls that lead up to the thrown exception. Call stacks provide valuable information for developers to locate and fix the cause of the crash. Figure 5.1 is an example of a call stack. In DAISY, we utilize call stacks to automatically collect the training set for our learning models.

5.2.2 Calling Context

A call graph is a directed graph, wherein app methods are nodes, and edges represent an invocation from a caller method to a callee method. The app's call graph is generated by statically analyzing the app code. Figure 5.2 shows a sample call graph of a login Activity for an Android app. When a user opens the app on their device, the login Activity will be launched. The `onCreate()`

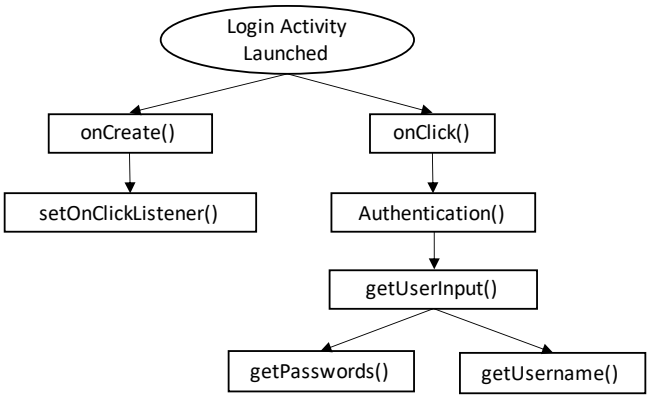


Figure 5.2: A sample call graph in activity.

callback will be invoked and an `onClickListener` is set to invoke `onClick()` when the user clicks (or taps) the "Login" button on the screen. Once the user clicks the button, `onClick()` will be invoked. It will then call `Authentication()` and `getUserInput()` methods, which will invoke `getPasswords()` or `getUsername()` to collect a username or a password respectively.

A calling context is a single path in the call graph which represents a method execution sequence. In Figure 5.2, the path from `onClick()` to `getPassword()` is a static trace, and the path from `onClick()` to `getUsername()` is another static trace. A method is *sensitive*, if it extracts sensitive information from the application or device. The methods `getUsername()` and `getPassword()` are sensitive, because they would return the username and password of the user. In this work, we propose DAISY to predict sensitive methods in the app's call graph.

5.3 Approach

Our approach aims to identify sensitive methods defined in unseen Android apps and third-party libraries. To achieve this goal, DAISY utilizes call stacks collected by dynamic analysis to created a large automatically labelled data set and then uses the data set to train classification models. For a given unclassified method with its calling context extracted from a call graph, the classification models will predict whether it may return sensitive information. Figure 5.3 shows the overview of our approach. For the data preparation shown on the left side of the figure, we

describe two important components (bold in the figure). The first component automatically labels data sets for training and the second component converts call stacks (during training) and static calling contexts (during prediction) into a format that can be processed and learned by a machine learning model. These processes result in artifacts which are used to train machine learning models as seen on the right side of the figure.

5.3.1 Training Data Preparation

We collect the call stacks and label them based on value-based dynamic taint analysis which identifies information flows by checking whether run-time values of variables contain predefined sensitive values (e.g., username). We use value-based dynamic taint analysis because it is robust enough to handle native code and out-of-scope data flows (e.g., information from third-party services such as Facebook, manual / web-based registration information from remote servers, information from file / user interface). The major limitation of value-based dynamic taint analysis is its inability to handle encrypted / obfuscated data, but since we are monitoring variables inside app code, we believe such inability may not cause much noises as encryption and obfuscation are typically performed only when data is sent out. The automatic labeling process consists of five steps as we introduce below.

Sensitive Data User Profile

To reduce noise in value-based dynamic taint analysis, sources must be unique, predefined values are so that other values do not accidentally contain them. We consider such values as the user profile of an Android app. The profile includes six unique identifiers device ID, serial number, Android ID, advertising ID, email address and user name. Table 5.2 shows example values of a user profile. Many apps collect and access profile data, which ensures this step is broadly applicable across a large number of apps. We chose those six information type because they are personally identifiable information (PII) defined by the EU General Data Protection Regulation (GDPR) [29, 31] and prior works [137] [220] also consider them as the major types of sensitive

data in privacy protection. This profile, labeled **Sensitive Data User Profile** in Figure 5.3, serves as an input for the auto-labeling component.

App Instrumentation

In order to identify the return value of `String`-type methods at run time, we instrumented all `String`-type methods in the smali code by inserting Android logging invocations at their return statements (**Instrument String Methods** in Figure 5.3). Each inserted invocation prints the run-time value of the returned variable and the call stack (i.e., run-time calling context) at the return statement. In particular, we acquire the call stack by throwing an exception and catching it immediately, while fetching the call stack saved in the exception variable. It should be noted we instrument only `String`-type methods because their return values are readable, and although sensitive data are often packaged into objects, they are typically accessed through `String`-type methods. For example, a method `getUserProfile()` may return an object of type `UserProfile` which contains username and android id as fields, but the actual values of username and android id are typically still accessed through `UserProfile.getName()` and `UserProfile.getID()` which are both `String`-type methods. For this process, we use Apktool [208] to decompile APK files into smali code[1]. The resulting **Instrumented APKs** are then used as input to generate call stacks, as seen in Figure 5.3.

Call Stack Generation

The resulting instrumented code is rebuilt back into the APK format for automatic runtime testing (**Generate Call Stacks** in Figure 5.3). We use the Android Debug Bridge (adb) to automatically install the rebuilt apps onto our test device and run MONKEY [2] to perform the testing of apps. For each app, we automatically install, execute, test, uninstall, and save the system log into the local file system for later inspection. We manually created accounts using the user profile data to complete the login process for apps that require registration and login during testing. This en-

[1] Assembler for the dex format used by Dalvik

Table 5.2: Sensitive Data User Profile

Info Type	Value
Advertising Id	"fc1303d8-7fbb-44d8-8a68-a79ffac06fea"
AndroidID	"a54eccb914c21863"
Email	"********@gmail.com"
IMEI	"355458061189396"
Serial	"ZX1G22KHQK"
User Name	"******"

Table 5.3: System log of `String`-type method call stack

```
1  09-12 16:25:13.442 W System.err: java.lang.Exception: fc1303d8-7fbb-44d8-8a68-a79ffac06fea
2  09-12 16:25:13.443 W System.err: at com.facebook.internal.AttributionIdentifiers.getAndroidAdvertiserId
                                    (AttributionIdentifiers.java:1)
3  09-12 16:25:13.443 W System.err: at com.facebook.marketing.internal.RemoteConfigManager.run
                                    (RemoteConfigManager.java:5)
4  09-12 16:25:13.443 W System.err: at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1133)
5  09-12 16:25:13.443 W System.err: at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:607)
6  09-12 16:25:13.443 W System.err: at java.lang.Thread.run(Thread.java:761)
```

sures that DAISY will identify sensitive call stacks by searching for the values from the predefined profile data. Table 5.3 shows an example of a sensitive trace where Line 1 shows the return value, Line 2 shows the method that has been invoked (`com.facebook.internal.AttributionIdentifier.AdvertiserId()`), and the rest of lines show the call stack of this method.

Automatic Labeling

When run time testing is completed, the system log contains the collected call stacks of the *String*-type methods and their corresponding return `String` values under the context of that call stack (**App Methods with Context** in Figure 5.3). If a call stack's return value matches a sensitive information type's value in Table 5.2, the call stack will be labeled with that sensitive information type (**Auto Label** in Figure 5.3). Otherwise it will be labeled as "non-sensitive". In the example of Table 5.3, Line 2 shows the `String`-type method `com.facebook.internal.AttributionIdentifiers.getAndroidAdvertiserId()`. Line 1 shows its return `String` value "c1303d8-7fbb-44d8-8a68-a79ffac06fea", which matches with the information type "AdvertiserID" in our user profile. In

this example, the call stack (Line 2-6) will be labeled with the sensitive information type "AdvertisingID". Note that a call stack can be labeled with multiple sensitive information types if its return value contains multiple values in TABLE 5.2

As discussed in Section 5.1, some methods may return different values in different calling contexts. To address this challenge, we label call stacks up to the length being considered, and label a call stack d with a sensitive information type t only when all the observed call stacks with d as their prefix returns the sensitive information type t. As an example, consider three dynamically observed call stacks with method a at the top: [email]$a \leftarrow b \leftarrow c$, [email]$a \leftarrow b \leftarrow d$, and []$a \leftarrow c \leftarrow e$. Here the first two call stacks return the email address, and the third call stack returns an empty string, and we assume that there are not other call stacks with a at their top. In such a case, if we consider call stacks with length one, we will mark a as insensitive because not all call stacks starting with a returns email. If we consider call stacks with length two, we will mark $a \leftarrow b$ as sensitive with type email, and $a \leftarrow c$ as insensitive. Using this strategy, we can make sure that we can identify both unconditional sensitive methods and conditional sensitive methods together with their calling context.

Method Signature Preprocessing

Each method signature from the call stacks is converted into a list of words that can be processed by a natural language processing model (**Preprocess Signatures** in Figure 5.3). At level 1, we remove parameters from the end of the method name and split the method signature on the dot operator (.). Next, each part of the method signature is split on the punctuation allowed in Android method names (i.e., _ and *$*), followed by word-splits at capitalization changes (i.e., camel case boundaries) using a simple regular expression. Finally, all words are changed to lowercase. After preprocessing, for example, the method signature `android.location.Location.getLatitude()` becomes `[android, location, location, get, latitude]`. For contexts at successive levels (e.g., level 2, level 3, etc.), the above steps are applied to each method signature in order and then concatenated into one word list.

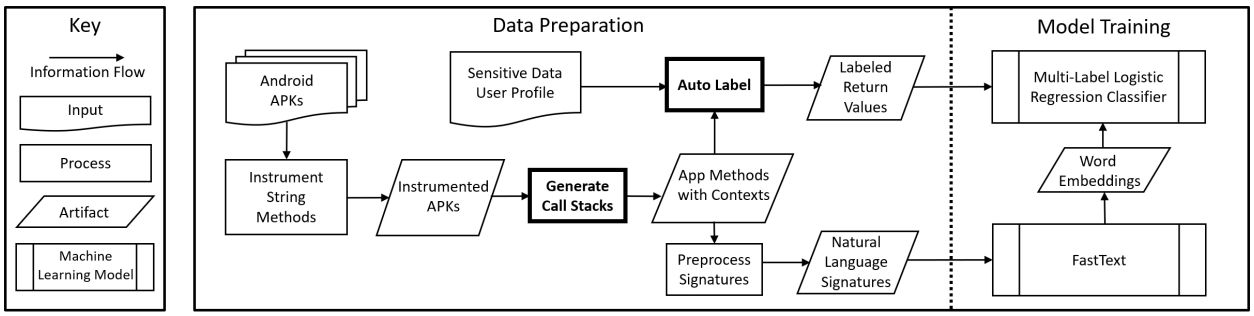


Figure 5.3: Model Training Approach Overview

As shown in Table 5.1, it is sometimes difficult to tell whether a method returns sensitive information by looking at the method itself. With this intuition, our approach predicts the information type of a method based on its calling context. In our approach, we consider different numbers of most recent calls, or trace length, in the call stack, for training and testing our model. This step consists of choosing the last call (level 1), the last two calls (level 2), and the last three calls (level 3) in each call stack.

5.3.2 Training of Classification Models

As described in Section 5.3.1, our approach produces **Labeled Return Values** and **Natural Language Signatures** as a result of our data preparation processes. In turn, these artifacts are used to train and configure two machine learning models: **FastText** and a **Mult-Label Logistic Regression Classifier** as seen in Figure 5.3.

Method Signature Word Embeddings

A calling context consists of a list of fully qualified method names denoting the order the methods to be called. These method names are usually composed of words from natural language (e.g., `android.location.Location.getLatitude()` is the method signature and it is pre-processed into a sequence of words: `[android, location, location, get, latitude]`. A common problem in word vector representations is how to handle *unknown words*. If a model has never seen a word previously, then finding a semantically relevant representation is difficult. *Unknown words* cause very severe problem in texts extracted from calling

contexts and call stacks due to the informal language usage in source code so multiple abbreviated words are often combined to form a code identifier (e.g., “addr” for “address”, “droid” for “Android”).

To overcome this challenge, we use FASTTEXT [130] to produce vector representations for the word sequence of a calling context. FASTTEXT is a text classification technique based on the skip-gram Word2Vec [150] model for learning vector representations for words and text classification. FASTTEXT framework helps to overcome the challenge of unknown words by breaking all words down into subwords, which are the set of overlapping character n-grams contained in a word. A character n-gram is simply a sequence of n characters of a word. Each character n-gram is assigned a vector representation and learned similar to a whole word. The subword vector representations and the whole word vector representation, if previously seen, are used to compute a final word representation. Thus, any new word representation can be approximated using its constituent subwords. For example, an unseen word “biometric” would have no existing vector representation, yet the tri-gram subwords would consist of:

```
<bi, bio, iom, ome, met, etr, tri, ric, ic>
```

Because subwords are shared across multiple words, they have a higher probability of being seen that whole words, and thus a higher probability of having existing vector representations. The subword vector representations are then averaged together to derive a semantically relevant whole word vector representation for “biometric”. Notably, in FastText, whole words are preprocessed by enclosing the words in the“<”and “>” characters to differentiate n-grams from n-length words, e.g., the tri-gram “met” can be differentiated from “<met>”.

It should be noted that although sub-word features are being considered, our data pre-processing (see 5.3.1) to split full method signatures to word sequence is still required so that sub-word extraction will be performed only within words.

Sensitive Method Classifier

The learning task is to classify a given method on whether its top method's return value is of a sensitive information type. Normally, one would construct a multi-class learning model that predicts the best information type class, among a few information type classes, plus one non-sensitive class. This approach has a limitation of multi-labeling. The multi-label problem arises because a single method can return multiple sensitive information types, yet multi-class models typically predict one best class for each input. To mitigate these limitations, we decompose the learning task into separate binary learning problems: for each sensitive information type, a binary classification predicts whether the type is a specific sensitive type or "not that type". If a given method is classified by all models as "not that type", then it is assumed to be non-sensitive.

After FASTTEXT has completed it's subword tokenization and embedding averaging (described in Section 5.3.2), the input vector representation is provided to six different binary logistic regression classifiers. The output of each classifier for a single method follows the equation below:

$$f(CallTrace) = g(B \sum_{i \in \Phi}^{m} AX_i/m) \tag{5.1}$$

$$B \in \mathbb{R}^{c \times h}, A \in \mathbb{R}^{h \times v}, X_i \in \mathbb{R}^v$$

where g is the sigmoid function, A is an embedding matrix containing all of the model's whole-word and subword embeddings, Φ is the set of all embedding indices that compose the full call trace representation, and X_i is a one-hot vector indicating the current index of the representation being summed. The matrix B holds the parameters for the logistic regression unit for each class. c, h,and v represent the number of classes, the hidden size of each embedding vector, and size of the vocabulary (full-word and subword vocabularies together), respectively. For clarity, the summation simply describes the averaging of the full-word and subword embeddings.

Both matrix A and B are randomly initialized, which means the method representations are completely learned during the training phase. We found in our experiments that initializing with

embeddings pre-trained on our call stack corpus did not improve classification performance for our task.

During training, labels from dynamic testing are given to the model to calculate the loss for each call stack. Each of the six regression unit's losses are calculated separately according to the well-known binary logistic regression loss function:

$$-y_{ic} \cdot log(f(CallStack)) + (1 - y_{ic}) \cdot log(f(CallStack)) \tag{5.2}$$

where $y_{ic} \in \{0, 1\}$ is an indicator variable for the i^{th} method's membership in class c. Minimizing the sum of equation 5.2 over each call stack is solved asynchronously using stochastic gradient descent [130]. Multiple threads are used to optimize performance and the number of threads is a hyperparameter of our model.

During inference, the output of equation 5.3.2 is calculated for each class. Our model predicts class membership using a threshold of 0.5 and, therefore, multiple classes can be predicted for a method. If no class probability is above 0.5 then the call trace is predicted to be "non-sensitive".

5.3.3 Collecting Methods for Prediction

After the classification models are trained, given an Android app, DAISY automatically extracts methods with their calling contexts and identify sensitive methods from them. Utilizing the Androguard framework [106], DAISY disassembles an app and extracts its method call graph. From the method call graph, DAISY extracts only `String`-type methods as *methods for prediction* as they may directly return sensitive information (for the same reason as explained in Section 5.3.1). It should be noted that other methods may still be a part of the calling contexts of these `String`-type methods. Then, for each method for prediction, DAISY traverses the call graph starting from its node and traverse backward towards the root (i.e., the program entry method) along all paths for up to two edges. By doing so, we are able to extract a list of static calling contexts for each method for prediction. They are pre-processed in the same way as described in Section 5.3.1, and then feed into the trained classification model. It should be noted that the

same method with different calling contexts and different level of calling contexts are fed into the classification models separately. For example, a method a with two calling contexts $a \leftarrow b \leftarrow c$ and $a \leftarrow b \leftarrow d$ will be separated to four inputs: a, $a \leftarrow b$, $a \leftarrow b \leftarrow c$, and $a \leftarrow b \leftarrow c$.

5.4 Evaluation

5.4.1 Research Questions

Our evaluation aims to answer the following research questions:

- RQ1: How precisely can DAISY identify sensitive methods and categorize them?
- RQ2: How many of the sensitive methods discovered by DAISY can not be found by applying existing static analysis and dynamic analysis?
- RQ3: How many additional conditional sensitive methods can DAISY find by considering more levels of calling contexts?

5.4.2 Experiment Process

From the call graphs of the 100 testing apps, Using the approach described in Section 5.3.3, we extracted 196,282 methods (i.e., considered as in-context methods with calling context to level 1), 904,476 in-context methods with calling contexts to level 2, and 1,715,993 in-context methods with calling contexts to level 3. We consider calling contexts up to level 3 because longer calling contexts will lead to even more data, leading to huge cost in the prediction process. Please note that in this paper our goal is to validate the usefulness of calling contexts, and we leave the determination of an optimal calling-context length (which may differ for different information types and apps) as future work. Meanwhile, we trained and validated 18 classification models (the combination of three calling context lengths and six information types) using the call stacks collected from apps in the training and validation set.

After that, we fed the collected in-context methods to their corresponding classification models (e.g., an in-context method with calling context to level 2 will be fed to all six classification models

trained / validated with call stacks with length 2). After the prediction process, DAISY reported 2,237, 7,214, and 17,476 in-context methods as sensitive for calling context level 1, 2, and 3, respectively.

5.4.3 Ground Truth Labelling

The 100 apps in our testing set contain more than 2.8 million in-context methods. So, it is impossible to label all of them and calculate the recall of DAISY. Furthermore, the recall metrics of DAISY does not affect its practical usability much. Since there is no approach that can exhaustively detect all sensitive methods anyway, any approach that can discover many more sensitive methods undetected by existing approaches or lists will provide much additional value. Since the users may need to either manually review the discovered sensitive methods, or manually review the detect information leaks from the discovered sensitive methods, the precision of DAISY can be very important, because it indirectly measures the additional effort DAISY users may need to spend when using it.

To calculate the precision of DAISY, we need to label the sensitive in-context methods reported to confirm true positives and false positives. However, it is still infeasible to label the entire set of more than 26,000 reported sensitive in-context methods. Therefore, we label the following two sampled subsets to evaluate DAISY in two different usage scenarios. In the first usage scenario, we consider DAISY to be used in the app scanning at app stores / security analysis services where a human user cannot review too many sensitive methods because she / he is handling a large number of apps. Therefore, the user may review only reported sensitive in-context methods that are most likely to be true positive. Therefore, we sample the first subset (high-confidence set) by choosing the 10 reported sensitive in-context methods with highest confidence from each of the 18 classification models (combination of three different calling-context lengths and six information types). In total, this subset contains 170 reported sensitive in-context methods (no positive in-context methods were reported for IMEI information type with calling context to level 1). In the second usage scenario, we consider DAISY to be used by the app developer / app producing organization

Table 5.4: Precision of DAISY on the High-Confidence Subset

InfoType	Level 1	Level 2	Level 3	Overall
AdsID	8/10	10 /10	8/10	26/30 (86.7%)
AndroidID	8/10	8/10	8/10	24/30 (80.0%)
Email	7/10	7/10	6/10	20/30 (66.7%)
IMEI	0/0	3/10	7/10	10/20 (50.0%)
Seriel	8/10	9/10	10/10	27/30 (90.0%)
Username	4/10	8/10	9/10	21/30 (70.0%)
Overall	35/50	45/60	48/60	128/170 (75.3%)

who wants to avoid privacy policy violations, so that they want to review all suspicious methods. For this scenario, we randomly chose 20% reported sensitive in-context methods from the top 5 ranked apps in the testing set. Using the second subset (random set), we can evaluate DAISY's precision on each of these apps. We manually labelled all the total 324 in-context methods in the two data sets. To reduce potential labelling errors, we have two authors to independently label all the methods, and a third author to resolve the conflicts. It should be noted that method labelling is a costly task which requires examination of all relevant code of the method in byte-code form, especially considering that much code are obfuscated in the top apps. Among the 314 in-context methods, there are still 22 that we cannot confirm whether they are really sensitive or not, so we conservatively considered all of them to be false positives in the evaluation.

5.4.4 Precision of Sensitive Method Discovery

To answer RQ1, we present the precision of DAISY on the high-confidence subset and the random subset in TABLE 5.4 and TABLE 5.5, respectively. In each table, Column 1 presents the information type (TABLE 5.4) or the app name (TABLE 5.5). To fit the table into the text column we use abbreviated package name of apps in TABLE 5.5 and the full names are in our anonymized website. Columns 2-4 present the precision for each information type / app with each length of calling context. In each cell, the number before the / symbol is the number of manually confirmed true positives, and the number after the / symbol is the number of reported in-context methods. Column 5 shows the overall precision which combines data in Columns 2-4.

Table 5.5: Precision of DAISY on the Random Subset

App	Level 1	Level 2	Level 3	Overall
simplygood.ct	1/2	3/5	6/13	10/20 (50.0%)
fitnessflow	0/1	1/5	2/6	3/12 (25.0%)
zengatv	1/1	5/12	5/22	11/35 (31.4%)
mlssoccer	3/5	10/14	15/24	28/43 (65.1%)
oki.letters	0/3	2/4	5/27	7/34 (20.6%)
Overall	5/12	21/40	33/92	59/144 (41.0%)

From TABLE 5.4, we can see that in the high-confidence subset, DAISY is able to achieve an average precision of 70%, 75% and 80% for calling context length one, two, and three, respectively. The overall precision of 75.3% indicate that the majority of in-context methods identified by DAISY are true positives, and a user will be able to identify 128 real sensitive in-context methods from 100 testing apps by reviewing only 170 top reported in-context methods. It should be noted that these‘170 in-context methods are from 41 different apps so they cover a large portion of testing apps. Among different information types, DAISY performs best on Android ID, Advertisement ID, and Serial Number, partly because they are more technical with standard names, so methods returning these information types tend to have more standard name. In contrast, Email and User Name are more difficult to identify due to the various ways to refer to them, but DAISY still achieved 66.7% precision for email and 70.0% precision for user name, indicating that using DAISY developers do not need to spend too much time on ruling out false positives. DAISY performs worst on IMEI, partly because methods returning IMEI are rarely called so the dataset becomes very unbalanced. In the training set, less than 0.1% of in-context methods are labeled with IMEI, indicating that a random selection will lead to a precision of only 0.1% and DAISY is 500 times more discriminative than that.

From TABLE 5.5, we can see that in the random subset, DAISY is able to achieve an average precision values from 20.6% to 65.1% in five apps, and an overall precision of 41.0%. From different calling-context lengths, the precision value is 5/12 (41.7%), 21/40 (52.5%), and 33/92 (35.9%), respectively. It should be noted that sensitive in-context methods are very sparse so the data unbalance is severe. In our training set, less than 1% of all in-context methods are labeled as

sensitive, indicating that a random selection will lead to an overall precision of less than 1%, and DAISY performs tens of times better than that.

5.4.5 Supplementing Existing Approaches

To answer RQ2, we check whether the sensitive in-context methods detected by DAISY can be detected by either static taint analysis or dynamic analysis.

Static Taint Analysis. Based on an existing list of sensitive API methods, it is possible to use static taint analysis to expand the list and discover more sensitive API methods. LIST 5.1 shows an example, where *sensitive method* `findDeviceId()` (detected by DAISY) obtained the IMEI information from Android API `android.telephony.TelephonyManager.getDeviceId()`, so `findDeviceId()` is considered to be detectable by static taint analysis. To check whether our reported in-context methods are also detectable by static taint analysis, we applied FlowDroid [79] using sensitive API method list of SUSI [161] as sources, and checks whether sensitive information may flow to the confirmed true positives in-context methods in TABLE 5.4 and TABLE 5.5. The analysis results show that none of the discovered in-context methods are detected with static taint analysis.

```
1 /* findDeviceId() is the source identified by DAISY */
2
3 /* android.telephony.TelephonyManager.getDeviceId() is an Android API source identified
      by Susi */
4
5 public String findDeviceId(Context context){
6     TelephonyManager tm = (TelephonyManager) context.getSystemService("phone");
7     return tm.getDeviceId();
8 }
```

Listing 5.1: An example of a DAISY source obtaining IMEI from a SUSI source

Dynamic Analysis. It is also possible to use dynamic analysis to detect sensitive methods. Actually, our automatic labelling in the training process can be deemed as an approach to discover sensitive methods. However, fully automatic dynamic analysis are limited by test coverage, while manual testing to improve test coverage will bring in additional human effort. In our comparison, we use MONKEY to perform the testing because it is fully automatic and robust enough to be applied to all apps. Furthermore, existing studies [200] show that it achieves comparable coverage

with state-of-art tools. For each app, we ran MONKEY for one hour without any human interaction and used automatic labelling described in Section 5.3.1 to identify sensitive in-context methods. The comparison shows that among the 187 true positives from TABLE 5.4 and TABLE 5.5, only nine are discovered with dynamic analysis. Eight of them (four for Ads ID, one for Android ID, and three for Serial Number) are from TABLE 5.4, and the remaining one is from app "simplely-good.ct" in TABLE 5.5.

To sum up, the comparison shows that almost all sensitive methods detected by DAISY are new and cannot be easily detected by static taint analysis or dynamic analysis.

5.4.6 Conditional Sensitive Methods

To answer RQ3, we performed a statistical analysis on the result to see how many extra sensitive in-context methods are identified when we increase the length of calling context to be considered. Note that when the considered length is equal to one, our model identifies only unconditional source methods. When the length increase, our model identifies more conditional source methods, together with their calling contexts.

Table 5.6 shows the additional sensitive in-context methods reported when increasing levels of calling contexts (numbers in Columns 3 and 4). From the table, we can see that increasing calling context level to two helps DAISY to report 4,195 more potential sensitive in-context methods, and increasing the calling context level to three further helps DAISY to report 9,324 more potential sensitive in-context methods. Note that an in-context method $a \rightarrow b$ is considered additional if and only if $a \rightarrow b$ is reported as sensitive in Level 2, but a is not reported as sensitive in Level 1. Table 5.7 shows the total additional sensitive in-context methods that are confirmed in our evaluation subsets. With calling-context length increased to two, DAISY identified 24 additional sensitive in-context methods that cannot be identified at level 1, increasing the calling-context length to three further helps to identify 12 sensitive in-context methods. Since in-context methods for different levels are sampled separately, in this table, an in-context method $a \rightarrow b$ is considered additional if and only if $a \rightarrow b$ is reported and manually confirmed as sensitive in Level 2, but a is

Table 5.6: Additionally Reported In-context Methods by Increasing Calling-Context Levels

InfoType	Level 1	Δ Level 2	Δ Level 3
Advertising ID	1128	1602	2368
Android ID	484	874	1132
Email	386	910	3986
IMEI	0	183	308
Serial	108	47	97
User name	131	569	1433
Total	**2237**	**4195**	**9324**

Table 5.7: Additionally Confirmed In-context Methods by Increasing Calling-Context Levels

InfoType	Level 1	Δ Level 2	Δ Level 3
Advertising ID	12	2	0
Android ID	8	3	1
Email	8	5	7
IMEI	0	3	4
Serial	8	3	0
User name	4	8	0
Total	**40**	**24**	**12**

not reported as sensitive in Level 1 (note that a does not need to be in the sampled sets). From the results in two tables, we can see that (1) increasing calling context length to two and three helps DAISY to detect many more in-context methods which are not detectable by classifying methods only and (2) the additionally detected in-context methods cover all information types, indicating that conditional sensitive methods are common in all information types.

5.4.7 Origin of DAISY sources

We further investigated the source methods DAISY identified to find out where the methods are declared. Most of the extra sensitive methods found by DAISY are from third-party libraries. For example, one Advertising ID source `com.facebook.internal.Attribution Identifiers.getAndroidAdvertiserId()` is from a Facebook library. For the sources that come from a user's input, like email and username, most sources were located within the app itself (i.e., app-specific methods). Such as an Email source `"com.dozuki.ifixit`

Return Value: **********@gmail.com	conditional source
Call Stack:	
com.crashlytics.android.core.CrashlyticsCore.sanitizeAttribute(CrashlyticsCore.java:845)	
com.crashlytics.android.core.CrashlyticsCore.setUserEmail(CrashlyticsCore.java:528)	
com.fitradio.ui.login.task.BaseLoginJob.handleLoginResponse(BaseLoginJob.java:146)	
com.fitradio.ui.login.task.EmailLoginJob.getUserLoginEvent(EmailLoginJob.java:68)	
com.fitradio.ui.login.task.BaseLoginJob.onRunRun(BaseLoginJob.java:67)	
Return Value: “expiration_date”	not source
Call Stack:	
com.crashlytics.android.core.CrashlyticsCore.sanitizeAttribute(CrashlyticsCore.java:845)	
com.crashlytics.android.core.CrashlyticsCore.setString(CrashlyticsCore.java:560)	
com.fitradio.ui.login.task.BaseLoginJob.handleLoginResponse(BaseLoginJob.java:153)	
com.fitradio.ui.login.task.EmailLoginJob.getUserLoginEvent(EmailLoginJob.java:68)	
com.fitradio.ui.login.task.BaseLoginJob.onRunRun(BaseLoginJob.java:67)	

Figure 5.4: An Email Conditional Source and Its Non-sensitive Context

`.ui.auth.LoginFragment.getEmail()"` is an app-specific source method from the app "com.dozuki.ifixit". We can tell from the method signature this method collects email account while a user login.

In Figure 5.4, we present a conditional source method of information type email that is idenfied by DAISY. In particular, method `com.crashlytics.android.core.CrashlyticsCore.sanitize Attribute()` returns the user's email address when it was invoked from `com.crashlytics.android.co UserEmail()`, but returns a non-sensitive information when it was called by other method. More examples of DAISY sources can be found in Figure 5.5.

5.4.8 Examples of Identified Sources

We present some identified example sources in Figure 5.5, covering various device identifiers, email, and username. From the figure we can see that, DAISY can not only identify unconditional source methods such as `getAndroidId()`, but also identify conditional source methods (e.g., `getString()`) which return general values but will return sensitive information under certain calling context.

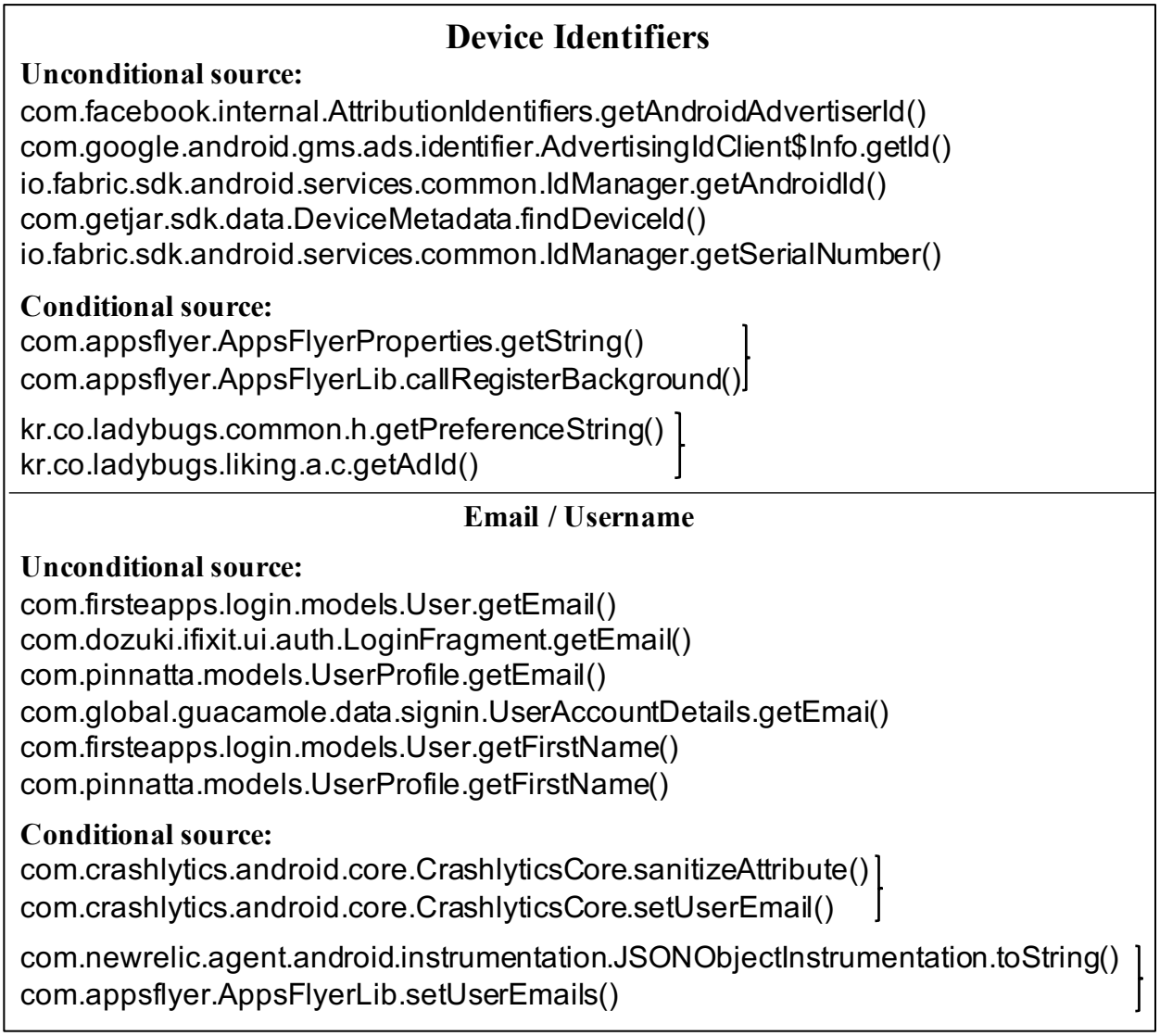


Figure 5.5: DAISY Sources

5.4.9 Threats to Validity

The main threat to the internal validity of our evaluation is in the selection and labeling of reported in-context methods. To reduce the bias and errors in the labelling process, we have two people to perform manual labelling independently, and a third people to resolve conflicts. The main threat to the external validity of our evaluation is that the results may apply to only the testing set and the labeled dataset. To provide a more comprehensive evaluation of DAISY, we used two different selection mechanisms, corresponding to two usage scenarios of DAISY. The evaluation results on two selected subsets are reasonably consistent as they both show high discrimination power of the classification models, but the results on the random subset is lower than those on the high-confidence subset, which is as expected.

5.5 Discussion

5.5.1 Using Conditional Sources

Once we identify the sensitive methods and determine their information type, the next step is to use them as sources in information flow analysis. For conditional sensitive methods, we need to make sure that their calling contexts are also encoded in the taint analysis. Some recent work [220] can encode the calling context as an information flow prefix in the IFDS [169] (Interprocedural Finite Distributive Subsets) static analysis framework, so that the calling context will be automatically matched in the information flow analysis (e.g., in FlowDroid).

5.5.2 Domain-Specific Information Types

In our research, we consider top Android apps and common sensitive information such as email address. However, our approach can also be applied to domain-specific information types such as transaction information in financial apps or grading information in education apps. To work on domain-specific information types, we can construct domain-specific app sets for sensitive application domains, such as education, finance, and health/medical care. Furthermore, to address

the potential sparsity problem for domain-specific information types, we plan to leverage transfer learning [214] which incorporates a model trained from a large general data set (e.g., general information type data set), and adjusts the weights and parameters of the model based on a smaller adaptation data set from the specific domain.

5.5.3 Evaluation Strategies

As it is to labor and time-intensive to manual verify and label all reported sensitive in-context methods, we manually validate a subset of the methods. We chose two approaches to select subsets. In the first selection strategy, we chose the 10 top most likely (i.e., highest probability) predictions made by DAISY for each of the six information types under each of the three context lengths. This resulted in 180 predictions for manual labeling and validation and represents the effectiveness of DAISY on the in-context methods that it is most confident are sensitive. This highest-confidence subset is useful to show DAISY's effectiveness for processing batches of apps which is of interest to third-party audits and quality and security assurance. For the second selection strategy, we selected five apps and randomly sampled 20% of DAISY predictions from those apps for manual labeling and validation. From the 100 apps in the testing set, we chose the top five apps, resulting in 144 in-context methods.

There were two other selection strategies we considered. The first was a confidence threshold strategy where separate evaluations were reported at different confidences (i.e., different prediction probability thresholds) to show the precision of DAISY at those confidences. Unfortunately, due to the size of the number of sensitive in-context predictions made it was infeasible to perform a manual labeling and validation of even the top confidence thresholds (i.e., 100% or 99% confidence). Instead, we chose the highest-confidence strategy in order to show DAISY's effectiveness at its very highest confidence. The second was a random sampling of the predictions. Again, due to the number of sensitive in-context predictions made it would be infeasible to attempt a manual labeling and validation of more than 1% of the total predictions. This sampling would be too small to guarantee a good representation of all ranks of apps throughout the dataset. We chose to perform

the ranked selection strategy instead in order to have a more accurate representation of the dataset.

Chapter 6: CONCLUSION AND FUTURE WORK

In this chapter, we summarize the contributions of this thesis and point out some directions for future work.

6.1 Thesis Summary

The main purpose of this thesis work is to uncover sensitive data access in mobile apps and reduce false negatives of taint analysis for privacy leak detection. More specifically, this thesis made significant contributions in the following aspects.

Firstly, this thesis presents an empirical study of how mobile apps collect and share user's data. We specifically focused on analyzing what data was collected and shared with analytic services, which is widely used to analyze users' behavior. We developed a semi-automatic framework to monitor data sent to analytic services during runtime. According to the findings of this study, many popular apps shared user's PII with analytic services, which may be a violation of their own privacy policies or analytic services' Terms of Service. This study provides a foundation for us to understand the data leakage from apps to third-party services.

Secondly, this thesis contribute to privacy leak detection by reducing false negatives in taint analysis caused by inaccessible code and incomplete source list. We developed a hybrid taint analysis framework to identify sensitive data access through methods defined by app and third-party services, and use them as additional sources in static taint analysis. We tested our framework on both benchmark and real-world apps from google play store. The evaluation results show that our new proposed framework effectively detects data access as well as privacy leaks that missed by the state-of-art static taint analysis tools.

Our previous proposed approach requires human effort to trigger sensitive data-related events as many as possible. To avoid human effort and test coverage limitation, we further proposed a machine learning based approach to identify sensitive data access through methods defined by app and third-party libraries. The classifier is trained on an automatically labelled data set of methods

and their calling context. We tested it on previously unseen apps and discovered that it was able to identify sensitive methods in them with high precision. The identified sensitive methods were used as sources in static taint analyses and detected privacy leaks that were missed by existing tools.

Although this thesis focused primarily on free Android apps in Google Play, we believe that the frameworks we developed could be applied to other App Markets such as the Amazon App Store or the Apple App Store. We expect that the knowledge we discovered about apps and the lessons we learned about data collection and leaks will assist users in making informed decisions, app developers in avoiding regulatory violations, and market owners in improving their current privacy frameworks.

Meanwhile, we acknowledge that privacy has many facets. This thesis only proposed a few possible ways to eliminate this problem. We believe other aspects, such as educating users and app developers, as well as improving and enforcing laws and regulations, are also critical for protecting the privacy of mobile users.

6.2 Future work

This thesis work also leaves several directions worth improving and extending.

6.2.1 Increase test coverage

Our approach to identifying sources is heavily reliant on UI testing. We intend to trigger more user data-related events in order to identify more sources. Current approaches of automatic UI testing suffer from low coverage, especially for the apps that require user input and communication with remote servers, such as the process of registration, login processes. We intend to explore techniques for generating appropriate user input or bypassing some communication by synthesizing the response data from the remote server when necessary.

6.2.2 Accountable software under privacy laws

There has been an increase in the number of laws and regulations designed to protect user's privacy, and software is expected to comply with these regulations. We intend to work towards a deeper understanding and formalization of the relationship between softwares and privacy laws. To assist developers in designing accountable software under those regulations, we could enforce the notice of data collection and sharing at the code level so that users are aware of what data is collected and with whom that data is shared.Furthermore, privacy laws require that each app to provide a privacy policy that describes data collection and sharing. We could use our code analysis frameworks to generate natural language texts that accurately reflect the app's behavior.

6.2.3 Privacy in Augmented Reality Application

Augmented Reality (AR) has gained increased public attention in recent years. Different from traditional apps, AR apps often have access to a much larger interface of the users' privacy, and the private data can be in a variety of formats, such as text, image, and video. It will be interesting to see how such complicated data affects user privacy. For example, an AR app often requires continuous access to the smartphone's camera. If an app is collecting more than necessary information from the users' camera, or if the data collection is not mentioned in the app's privacy policy, users' privacy is compromised.

BIBLIOGRAPHY

[1] Flowdroid default source and sinks. https://github.com/secure-software-engineering/FlowDroid/blob/master/soot-infoflow-android/SourcesAndSinks.txt.

[2] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html, 2012.

[3] Didfail. https://www.cert.org/secure-coding/tools/didfail.cfm, 2015.

[4] Droidsafe. https://mit-pac.github.io/droidsafe-src/, 2015.

[5] Iccta. https://sites.google.com/site/icctawebpage/source-and-usage, 2016.

[6] Amandroid. https://bintray.com/arguslab/maven/argus-saf/3.1.2, 2017.

[7] Daildroid. https://github.com/dialdroid-android/DIALDroid, 2017.

[8] Equifax data breach. http://fortune.com/2018/09/07/equifax-data-breach-one-year-anniversary/, 2017.

[9] Add Firebase to an app. https://firebase.google.com/docs/android/setup, 2018.

[10] Appbrain Android analytics libraries. https://www.appbrain.com/stats/libraries/tag/analytics/android-analytics-libraries, 2018.

[11] Appbrain, firebase. https://www.appbrain.com/stats/libraries/details/firebase/firebase, 2018.

[12] AppsFlyer provide encryption option in API setUserEmails. "https://support.appsflyer.com/hc/en-us/articles/207032126-AppsFlyer-SDK-Integration-Android, 2018.

[13] Coverity. https://scan.coverity.com/, 2018.

[14] Crashlytics. https://docs.fabric.io/android/crashlytics/enhanced-reports.html, 2018.

[15] Crashlytics dashboard. https://stackoverflow.com/questions/34888420/crashlytics-how-to-see-user-name-email-id-in-crash-details/, 2018.

[16] Distribution dashboard about android platform versions. https://developer.android.com/about/dashboards/, 2018.

[17] Firebase collect device identifier. https://support.google.com/firebase/answer/6318039?hl=en, 2018.

[18] Firebase collect user event by default. https://support.google.com/firebase/answer/6317485?hl=en&ref_topic=6317484, 2018.

[19] Firebase collect user info by default. https://support.google.com/firebase/answer/6317486?hl=en&ref_topic=6317484, 2018.

[20] Firebase dashboard sample. https://www.bounteous.com/insights/2017/10/12/google-analytics-firebase-reporting-introduction/, 2018.

[21] Firebase log events. https://firebase.google.com/docs/analytics/android/events, 2018.

[22] Firebase set user ID. https://firebase.google.com/docs/analytics/userid, 2018.

[23] Firebase set user preoperties. https://firebase.google.com/docs/analytics/android/properties, 2018.

[24] Firebase user propertise. https://support.google.com/firebase/answer/6317519?hl=en&ref_topic=6317489, 2018.

[25] Flurry api setuserid(). https://developer.yahoo.com/flurry/docs/analytics/gettingstarted/technicalquickstart/android/, 2018.

[26] Flurry dashboard. https://developer.yahoo.com/flurry/docs/analytics/lexicon/eventreporting/, 2018.

[27] Ftc imposes $5 billion penalty and sweeping new privacy restrictions on facebook. https://www.ftc.gov/news-events/press-releases/2019/07/ftc-imposes-5-billion-penalty-sweeping-new-privacy-restrictions, 2018.

[28] GDPR anonymous data. https://gdpr-info.eu/recitals/no-26/, 2018.

[29] GDPR definition of personal data. https://gdpr-info.eu/art-4-gdpr/, 2018.

[30] GDPR lawfulness of processing. https://gdpr-info.eu/art-6-gdpr/, 2018.

[31] GDPR online identifiers for profiling and identification. https://gdpr-info.eu/recitals/no-30/, 2018.

[32] Google Advertising ID. http://www.androiddocs.com/google/play-services/id.html, 2018.

[33] Google play downloader via command line. https://github.com/matlink/gplaycli, 2018.

[34] Google+ shutdown. https://developers.google.com/+/integrations-shutdown, 2018.

[35] Google's definition of Android ID. https://developer.android.com/reference/android/provider/Settings.Secure#ANDROID_ID, 2018.

[36] IronSource recommend google advertising id. https://developers.ironsrc.com/ironsource-mobile/android/advanced-settings/#step-2, 2018.

[37] Isonsource collect user info by default. https://developers.ironsrc.com/ironsource-mobile/android/advanced-settings/#step-3, 2018.

[38] Marriott data breach. https://www.consumer.ftc.gov/blog/2018/12/marriott-data-breach, 2018.

[39] The Marriott data breach. https://www.consumer.ftc.gov/blog/2018/12/marriott-data-breach, 2018.

[40] Mixpanel collect user event by default. https://help.mixpanel.com/hc/en-us/articles/115004596186-Which-common-mobile-events-can-Mixpanel-collect-on-my-behalf-automatically-, 2018.

[41] Mixpanel collect user info by default. https://help.mixpanel.com/hc/en-us/articles/115004613766-Default-Properties-Collected-by-Mixpanel, 2018.

[42] Mixpanel's rule about using API. https://help.mixpanel.com/hc/en-us/articles/360000679006-Managing-Personal-Information, 2018.

[43] Number of available applications in the google play. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2018.

[44] Number of mobile phone users worldwide. https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/, 2018.

[45] Playdron metadata. https://archive.org/details/android_apps&tab=about, 2018.

[46] Privacy policy of Crashlytics. https://try.crashlytics.com/terms/privacy-policy.pdf, 2018.

[47] Privacy policy of Key Ring. =https://keyringapp.com/privacy/, 2018.

[48] Privacy policy of MightyText. https://mightytext.net/privacy, 2018.

[49] Privacy policy of Money Lover. https://moneylover.me/policy.html, 2018.

[50] Privacy policy of NYTimes. https://help.nytimes.com/hc/en-us/articles/115014892108-Privacy-policy, 2018.

[51] Privacy policy of Raaga. = https://www.raaga.com/privacy, 2018.

[52] Privacy policy of Rosetta Stone. https://resources.rosettastone.com/CDN/us/agreements/US_Privacy_Policy-102513.pdf, 2018.

[53] Privacy policy of SnipSnap. http://www.snipsnap.it/privacy-policy/, 2018.

[54] Statement by the acting director of ftc's bureau of consumer protection regarding reported concerns about facebook privacy practices. March 2018.

[55] Statement by the acting director of ftc's bureau of consumer protection regarding reported concerns about facebook privacy practices. https://www.ftc.gov/news-events/press-releases/2018/03/

statement-acting-director-ftcs-bureau-consumer-protection, 2018.

[56] Universal Analytics usage guidelines. https://support.google.com/analytics/answer/2795983?hl=en, 2018.

[57] Market share of firebase. https://www.appbrain.com/stats/libraries/details/firebase/firebase, 2019.

[58] Market share of flurrt. https://www.appbrain.com/stats/libraries/details/flurry/flurry-analytics, 2019.

[59] Market share of google analytics. https://www.appbrain.com/stats/libraries/details/analytics/google-analytics, 2019.

[60] Market share of mixpanel. https://www.appbrain.com/stats/libraries/details/mixpanel/mixpanel, 2019.

[61] Privacy policy of emojidom. http://www.emojidom.com/privacy-policy, 2019.

[62] Privacy policy of Gallery. https://www.avast.com/privacy-policy, 2019.

[63] Privacy policy of Staples. https://www.staples.com/hc?id=dbb94c10-973c-478b-a078-00e58f66ba32, 2019.

[64] Privacy policy ofShopclues. http://m.shopclues.com/rules-and-policies.html, 2019.

[65] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html, 2019.

[66] Understanding pii in google's contracts and policies. https://support.google.com/analytics/answer/7686480?hl=en, 2019.

[67] Analyze a stack trace. https://developer.android.com/studio/debug/stacktraces, 2020.

[68] Tinder, grindr and other apps share sensitive personal data with advertisers. https://www.npr.org/2020/01/14/796427696/study-grindr-tindr-and-other-apps-share-sensitive-personal-data-with-a 2020.

[69] Top android os versions. https://www.appbrain.com/stats/top-android-sdk-versions, 2020.

[70] Understand the activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle, 2020.

[71] Cost of data breach. https://www.ponemon.org/, 2021.

[72] Number of android apps. https://www.appbrain.com/stats, 2021.

[73] Privacy policy of tiktok. https://www.tiktok.com/legal/privacy-policy, 2021.

[74] Whatsapp faces $267m fine for breaching europeâs gdpr. https://techcrunch.com/2021/09/02/whatsapp-faces-267m-fine-for-breaching-europes-gdpr/, 2021.

[75] Maqsood Ahmad, Valerio Costamagna, Bruno Crispo, and Francesco Bergadano. Teicc: targeted execution of inter-component communications in android. In *Proceedings of the symposium on applied computing*, pages 1747–1752, 2017.

[76] Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*, pages 378–400. Springer, 2013.

[77] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[78] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

[79] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[80] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[81] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

[82] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 43–59. IEEE, 2009.

[83] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22. ACM, 2011.

[84] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86âa platform for analyzing x86 executables. In *International Conference on Compiler Construction*, pages 250–254. Springer, 2005.

[85] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*, pages 49–54. ACM, 2011.

[86] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[87] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47. ACM, 2008.

[88] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[89] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

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

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

[92] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in java. *ACM SIGPLAN Notices*, 34(11):2–11, 1999.

[93] Mark Brodie, Sheng Ma, Leonid Rachevsky, and Jon Champlin. Automated problem determination using call-stack matching. *Journal of Network and Systems Management*, 13(2):219–237, 2005.

[94] Jonathan Burket, Lori Flynn, William Klieber, Jonathan Lim, Wei Shen, and William Snavely. Making didfail succeed: Enhancing the cert static taint analyzer for android app sets. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH United States, 2015.

[95] Terence Chen, Imdad Ullah, Mohamed Ali Kaafar, and Roksana Boreli. Information leakage through mobile analytics services. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 15. ACM, 2014.

[96] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.

[97] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 144–155. ACM, 2016.

[98] Ravi Chugh, Jeffrey A Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–62, 2009.

[99] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[100] Federal Trade Commission. Children's online privacy protection act of 1998 (coppa), 1998.

[101] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. 2017.

[102] Christoph Csallner and Yannis Smaragdakis. Check'n'crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering*, pages 422–431. ACM, 2005.

[103] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):8, 2008.

[104] Arjun Dasgupta, Vivek Narasayya, and Manoj Syamala. A static analysis framework for database applications. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1403–1414. IEEE, 2009.

[105] Mixpanel dashboard. `https://help.mixpanel.com/hc/en-us/articles/360000865566-Set-up-Your-Tracking/`, 2018.

[106] Anthony Desnos. Androguard. `https://github.com/androguard/androguard`, 2015.

[107] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 118–128, 2007.

[108] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, 2010.

[109] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[110] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

[111] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, pages 1–14, 2012.

[112] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *2003 Symposium on Security and Privacy, 2003.*, pages 62–75. IEEE, 2003.

[113] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587, 2014.

[114] Centers for Medicare & Medicaid Services. The health insurance portability and accountability act of 1996 (hipaa), 1996.

[115] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland,* `http://www.cs.umd.edu/avik/projects/scandroidascaa`, 2(3), 2009.

[116] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 283–300, New York, NY, USA, 2009. ACM.

[117] Susan E Gindin. Nobody reads your privacy policy or online contract: Lessons learned and questions raised by the ftc's action against sears. *Nw. J. Tech. & Intell. Prop.*, 8:1, 2009.

[118] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[119] Seungyeop Han, Jaeyeon Jung, and David Wetherall. A study of third-party tracking by mobile apps in the wild. *Univ. Washington, Tech. Rep. UW-CSE-12-03-01*, 2012.

[120] John Heaps, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. Toward detection of access control models from source code via word embedding. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, pages 103–112, 2019.

[121] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[122] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[123] Kevin J Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. *ACM Sigplan Notices*, 44(6):453–464, 2009.

[124] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[125] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, pages 104–111. IEEE, 2016.

[126] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. {SUPOR}: Precise and scalable sensitive user input detection for android apps. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 977–992, 2015.

[127] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1037–1049. ACM, 2017.

[128] Jim Isaak and Mina J Hanna. User data privacy: Facebook, cambridge analytica, and privacy protection. *Computer*, 51(8):56–59, 2018.

[129] Haojian Jin, Minyi Liu, Kevan Dodhia, Yuanchun Li, Gaurav Srivastava, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. Why are they collecting my data? inferring the purposes of network traffic in mobile apps. 2(4), 2018.

[130] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[131] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[132] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 279–288. ACM, 2008.

[133] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Canât live with âem, canât live without âem. In *International Conference on Information Systems Security*, pages 56–70. Springer, 2008.

[134] Monica S Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 3–12, 2008.

[135] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 218–232. IEEE, 2007.

[136] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A Schmidt. Automata-based confidentiality monitoring. In *Annual Asian Computing Science Conference*, pages 75–89. Springer, 2006.

[137] Christophe Leung, Jingjing Ren, David Choffnes, and Christo Wilson. Should you use the app for that? comparing the privacy implications of app-and web-based online services. In *Proceedings of the 2016 Internet Measurement Conference*, pages 365–372, 2016.

[138] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: De-

tecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[139] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329. ACM, 2016.

[140] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for java. In *European Conference on Object-Oriented Programming*, pages 27–53. Springer, 2014.

[141] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1):2–16, 2005.

[142] Xing Liu, Sencun Zhu, Wei Wang, and Jiqiang Liu. Alde: privacy risk analysis of analytics libraries in the android ecosystem. In *International Conference on Security and Privacy in Communication Systems*, pages 655–672. Springer, 2016.

[143] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[144] Benjamin Livshits, John Whaley, and Monica S Lam. Reflection analysis for java. In *Asian Symposium on Programming Languages and Systems*, pages 139–160. Springer, 2005.

[145] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 229–240, 2012.

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

[147] Kirsten Martin and Katie Shilton. Putting mobile application privacy in context: An empirical study of user privacy expectations for mobile devices. *The Information Society*, 32(3):200–216, 2016.

[148] Isabella Mastroeni and Damiano Zanardini. Data dependencies and program slicing: from syntax to abstract semantics. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 125–134. ACM, 2008.

[149] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. The price of free: Privacy leakage in personalized mobile in-apps ads. In *NDSS*, 2016.

[150] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[151] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 993–1008, 2015.

[152] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)(San Diego, California, USA*, 2018.

[153] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[154] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *ACM Sigplan Notices*, volume 43, pages 308–318. ACM, 2008.

[155] Jonathan A Obar and Anne Oeldorf-Hirsch. The biggest lie on the internet: Ignoring the privacy policies and terms of service policies of social networking services. *Information, Communication & Society*, 23(1):128–147, 2020.

[156] European Parliament and The Council of the European Union. General data protection regulation (gdpr), 2016.

[157] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341, 2018.

[158] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341, 2018.

[159] François Pottier and Vincent Simonet. Information flow inference for ml. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, 2002.

[160] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 135–148. IEEE, 2006.

[161] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125. Citeseer, 2014.

[162] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

[163] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. 2018.

[164] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. Haystack: In situ mobile traffic analysis in user space. *arXiv preprint arXiv:1510.01419*, pages 1–13, 2015.

[165] Joel R Reidenberg, Travis Breaux, Lorrie Faith Cranor, Brian French, Amanda Grannis, James T Graves, Fei Liu, Aleecia McDonald, Thomas B Norton, and Rohan Ramanath. Disagreeable privacy policies: Mismatches between meaning and users' understanding. *Berkeley Tech. LJ*, 30:39, 2015.

[166] Jingjing Ren, Martina Lindorfer, Daniel J. Dubois, Ashwin Rao, David R. Choffnes, and Narseo Vallina-Rodriguez. Bug fixes, improvements, ... and privacy leaks - A longitudinal study of PII leaks across android app versions. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[167] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 361–374. ACM, 2016.

[168] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

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

[170] Dustin Rhodes, Cormac Flanagan, and Stephen N Freund. Bigfoot: static check placement for dynamic race detection. *ACM SIGPLAN Notices*, 52(6):141–156, 2017.

[171] A Rountev, D Yan, S Yang, H Wu, Y Wang, and H Zhang. Gator: Program analysis toolkit for android, 2017.

[172] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[173] Alejandro Russo and Andrei Sabelfeld. Securing timeout instructions in web applications. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 92–106. IEEE, 2009.

[174] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[175] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[176] Suranga Seneviratne, Harini Kolamunna, and Aruna Seneviratne. A measurement study of tracking in paid mobile applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 7. ACM, 2015.

[177] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D Bond, and Milind Kulkarni. Hybrid static–dynamic analysis for statically bounded region serializability. In *ACM SIGPLAN Notices*, volume 50, pages 561–575. ACM, 2015.

[178] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

[179] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David A Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220. Citeseer, 2001.

[180] Shashi Shekhar, Michael Dietz, and Dan S Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, volume 2012, 2012.

[181] M Sheldon, GV Weissman, and B Retrace. Collecting execution trace with virtual machine deterministic replay [c]. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*, 2007.

[182] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 203–217. IEEE, 2007.

[183] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 25–36. ACM, 2016.

[184] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[185] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In *Asian Symposium on Programming Languages and Systems*, pages 485–503. Springer, 2015.

[186] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *NDSS*, 2016.

[187] Mingshen Sun, Tao Wei, and John Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342. ACM, 2016.

[188] Ruoxi Sun and Minhui Xue. Quality assessment of online automated privacy policy generators: an empirical study. In *Proceedings of the Evaluation and Assessment in Software Engineering*, pages 270–275. 2020.

[189] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *22nd Annual Network and Distributed System Security Symposium*, 2015.

[190] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *International Static Analysis Symposium*, pages 352–367. Springer, 2005.

[191] Andreas Thies and Eric Bodden. Refaflex: Safer refactorings for reflective java programs. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 1–11, 2012.

[192] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[193] Omer Tripp and Julia Rubin. A bayesian approach to privacy enforcement in smartphones. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 175–190, 2014.

[194] Connor Tumbleson and Ryszard WiÅŻniewski. Apktool-a tool for reverse engineering 3rd party, closed, binary android apps, 2017.

[195] Narseo Vallina-Rodriguez, Jay Shah, Alessandro Finamore, Yan Grunenberger, Konstantina Papagiannaki, Hamed Haddadi, and Jon Crowcroft. Breaking for commercials: characterizing mobile advertising. In *Proceedings of the 2012 Internet Measurement Conference*, pages 343–356. ACM, 2012.

[196] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 683–693, 2017.

[197] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 221–233, 2014.

[198] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis.

[199] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–276, 2000.

[200] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. An empirical study of android test generation tools in industrial cases. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 738–748. IEEE, 2018.

[201] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D Breaux, and Jianwei Niu. Guileak: Tracing privacy policy claims on user input data for android applications. In *Proceedings of the 40th International Conference on Software Engineering*, pages 37–47. ACM, 2018.

[202] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Still: Exploit code detection via static taint and initialization analyses. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 289–298. IEEE, 2008.

[203] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[204] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceed-*

ings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pages 1329–1341, 2014.

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

[206] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346, 2013.

[207] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE, 2015.

[208] R Winsniewski. Android–apktool: A tool for reverse engineering android apk files, 2012.

[209] Michelle Y Wong and David Lie. Tackling runtime-based obfuscation in android with {TIRO}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1247–1262, 2018.

[210] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*, pages 899–914. IEEE, 2015.

[211] Cong Xie, Wei Xu, and Klaus Mueller. A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. *IEEE transactions on visualization and computer graphics*, 25(1):215–224, 2018.

[212] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.

[213] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering*, pages 101–104, 2012.

[214] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[215] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. Taintman: an art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[216] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157. Springer, 2010.

[217] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can we trust the privacy policies of android apps? In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 538–549. IEEE, 2016.

[218] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. Toward automatically generating privacy policy for android apps. *IEEE Transactions on Information Forensics and Security*, 12(4):865–880, 2016.

[219] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 353–363. ACM, 2011.

[220] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. Condysta: Context-aware dynamic supplement to static taint analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 796–812. IEEE, 2021.

[221] Yifei Zhang, Yue Li, Tian Tan, and Jingling Xue. Ripple: Reflection analysis for android apps in incomplete information environments. *Software: Practice and Experience*, 48(8):1419–1437, 2018.

[222] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[223] Leah Zhao, Neil Wong Hon Chan, Shanchieh Jay Yang, and Roy W. Melton. Privacy sensitive resource access monitoring for android systems. In *2015 24th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, 2015.

[224] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 37–48, 2015.

[225] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104, 2012.

[226] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. Automated analysis of privacy requirements for mobile apps. In *Proceedings 2017 Network and Distributed System Security Symposium*, 2017.

VITA

Xueling Zhang was born and brought up in Sichuan, China. She earned her bachelor's degree in Computer Science from Chongqing University of Posts and Telecommunications (CQUPT), China. Before enrolling in the PhD program, she worked in different roles in software engineering for eight years in China. She enrolled in the PhD program at The University of Texas at San Antonio (UTSA) in Fall 2017. Her main interest is improving privacy leak detection in Android mobile applications. Some specific areas of interest are program analysis, taint analysis, privacy analysis, GUI testing, and machine learning. In Summer 2020, she worked at Pacific Northwest National Laboratory (PNNL) as a Research Intern. Due to her excellence in research, the UTSA computer science department awarded her the Outstanding Achievement in Research Award 2020.