DroidJust: Automated Functionality-Aware Privacy Leakage Analysis for Android Applications

Xin Chen
The Pennsylvania State University
University Park, PA, USA
xinchen@cse.psu.edu

Sencun Zhu
The Pennsylvania State University
University Park, PA, USA
szhu@cse.psu.edu

ABSTRACT

Android applications (apps for short) can send out users' sensitive information against users' intention. Based on the stats from Genome and Mobile-Sandboxing, 55.8% and 59.7% Android malware families feature privacy leakage. Prior approaches to detecting privacy leakage on smartphones primarily focused on the discovery of sensitive information flows. However, Android applications also send out users' sensitive information for legitimate functions.

Due to the fuzzy nature of the privacy leakage detection problem, we formulate it as a justification problem, which aims to justify if a sensitive information transmission in an app serves any purpose, either for intended functions of the app itself or for other related functions. This formulation makes the problem more distinct and objective, and therefore more feasible to solve than before. We propose DroidJust, an *automated* approach to justifying an app's sensitive information transmission by bridging the gap between the sensitive information transmission and application functions. We also implement a prototype of DroidJust and evaluate it over more than 6000 Google Play apps and more than 300 known malware collected from VirusTotal. Our experiments show that our tool can effectively and efficiently analyze Android apps for the purposes of their sensitive information flows, and hence can greatly assist in detecting privacy leakage.

Categories and Subject Descriptors

D 2.4 [**Software Engineering**]: Software/Program Verification; D 4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Android security, Privacy leakage detection, Static taint analysis

1. INTRODUCTION

Mobile devices, particularly smartphones and tablets, are becoming more and more prevalent in the world. While users enjoy the convenience and functions brought by smartphones and tablets, their privacy is severely threatened by malicious mobile apps that leak sensitive information to remote servers against users' intention. Based on the statistics from Genome [35] and Mobile-Sandbox [2], 55.8% and 59.7% Android malware families feature privacy leakage. Therefore, it is vital to have an effective approach for detecting such malicious apps.

Prior approaches to detecting privacy leakage on smartphones primarily focused on the discovery of sensitive information flows [5, 8–10, 12, 13, 16, 21, 26]. However, as more and more benign apps send out users' sensitive information for legitimate application functions, these approaches cannot easily justify the purposes of sensitive information transmissions in an app, and hence may not detect privacy leakage effectively. For example, Google Maps sends out users' location information to a remote server for driving navigation and location-based recommendation services. To continue to be effective and adapt to the growing application markets, the development of more advanced analysis approaches to detecting privacy leakage on smartphones is strongly desired.

In this work, we formulate the problem of sensitive information leakage as a justification problem, which aims to justify if a sensitive information transmission in an app serves any purpose, either for intended functions of the app itself or for other related functions such as advertisements and analytics. To solve the justification problem, we propose an automated approach, called DroidJust. DroidJust not only identifies sensitive information flows, but also tries to link each flow with certain application function to provide the evidence for justification. DroidJust uses various static taint analyses to automate the whole analysis process. We evaluate DroidJust on more than 6000 Google Play apps and more than 300 known malware collected from VirusTotal. Our experiments show that our tool can effectively and efficiently analyze Android apps for the purposes of their sensitive information flows, and hence can greatly assist in detecting privacy leakage.

The contribution of this paper is as follows:

- We propose a novel approach to automatically justify an app's sensitive information transmission by bridging the gap between the sensitive information trans-

mission and application functions. Different to the previous work that utilize the evidence arising before or at the release point [28, 32], we are (probably) the first to consider the evidence arising after the release point for privacy leakage detection.

- Our approach overcomes several challenges to integrate all three types of PScout [6] resources (API, URI and Intent Actions) into DroidJust for labeling almost all (if not all) sensitive information sources in Android (in Section 4.2).
- We implement a prototype of DroidJust and evaluate it with more than 6000 Google Play apps and more than 300 known malware featuring privacy leakage. Our evaluation results demonstrate that DroidJust can effectively distinguish benign apps delivering sensitive information for application functions from the malware harvesting users' sensitive information.
- DroidJust identified 15 Google play apps that send out users' sensitive information but not for any application functions. Most of them cannot be detected by any anti-virus engine in VirusTotal and are still available for download in Google Play.

The rest of this paper is organized as follows. Section 2 introduces the motivation of our work, problem statement and design goals. We give an overview of our approach and a motivating example in Section 3, followed by detailed system design in Section 4. Section 5 presents the evaluation of DroidJust and presents our results. We discuss the limitations and related work in Section 6 and 7. Finally, we conclude our work in Section 8.

2. PROBLEM STATEMENT AND DESIGN GOALS

Recently, detecting privacy leaks in mobile apps has been one of the main research focuses on smartphone security, and it has led to development of many useful tools such as TaintDroid [9] for Android and PiOS [8] for iOS. Based on either static or dynamic taint analysis, such tools [5, 8–10, 12, 13, 16, 21, 26] can help discover potential sensitive information transmission. In a nutshell, these taint analysis approaches reduce the privacy leakage detection problem to the reachability problem. However, in reality, the existence of sensitive information transmission is *not* equal to privacy leakage, as real-world apps may send out users' sensitive information for their advertised functions. For example, a weather forecast app may send out users' location information to fetch the weather reports tailored to the locations; Google Maps also sends out GPS information for driving navigation. While these examples demonstrate obvious reasons for usage of users' sensitive information, there are also less obvious, sometimes even unpredictable, usage cases. For example, `com.pixeltech.imonline`, a trial Facebook messager app identified in our experiment, sends out users' Gmail addresses to a remote server for calculating the remaining trial days and then shows the number of days in the app. Judging this sort of sensitive information transmission is beyond the power of the conventional taint analysis approaches.

Realizing the fuzzy nature of the privacy leakage detection problem, prior research work has tackled the privacy leakage detection problem from different angles. For example, Yang *et al.* [32] proposed to use users' expectations as the indicator of privacy leakage. If the sensitive information transmission is expected by users, it will be considered as necessary, so not a leakage case; otherwise, if unexpected, it will be a privacy leakage case. However, *users' expectations are diverse*. For example, an advertisement library may send out a phone's geographic location for location-based advertisements. Depending on whether they like to receive targeted advertisements or not, different users may agree or disagree that disclosure of location information is expected in this context. Further, *we cannot assume all users are capable of comprehending system-level contextual information to provide their expectations*. The experiment in [32] has demonstrated that even security specialists had discrepancy about the usage of device IDs in certain apps after they reviewed the generated event chains that lead to data transmission. This is because app developers could potentially use a device ID, a phone number or even a Google account as a unique identifier of a device or a user, and such code-level information is often not available to the human specialists when they make decisions. Indeed, device IDs and phone numbers are the most common sensitive information that are delivered to the network [9, 35]. Different from the users' expectation angle, Tripp *et al.* [28] formulated the privacy-leakage detection problem as a machine learning problem based on certain features. Their approach however is probabilistic, and the effectiveness highly depends on the selected features and the training data sets.

In this work, we take a slightly different angle to tackle this privacy leakage detection problem. We formulate it as a *justification* problem, which aims to justify if a sensitive information transmission in an app serves any purpose, either for intended functions of the app itself or for other related functions such as advertisements and analytics. For example, if an app sends out the user's location to a remote server, later receives information from the server, and finally displays the information to the user in the phone screen, we consider this sensitive information transmission *justifiable*. On the contrary, if the app does not receive any information from the server after sending out users' location information, this sensitive information transmission is *unjustifiable*. Note that conceptually there are differences between *justifiability* and *privacy leakage*. According to our definition, a sensitive information transmission caused by an advertisement library is also justifiable because it does serve some known purpose, although privacy advocators may dislike it and consider it a privacy leakage. The merit of our formulation is that it separates technical issues from users' opinions. Rather than directly telling a user whether a sensitive information flow is a privacy leakage, we only report the purpose it serves, if any. The research problem now becomes more distinct and objective, and therefore more feasible to solve than before.

Following the formulation above, we aim to design an approach to justifying an app's sensitive information transmission. Specifically, we want to achieve the following design goals.

- **Fully automated analysis.** The proposed approach must be able to automatically justify an app's sensitive information transmission. The purpose is to minimize the involvement of human analysts in the middle. This task is challenging because it requires automatically extracting and understanding the contextual information in order

to bridge the gap between an app's sensitive information transmissions and its functions.

- **Complete and precise coverage.** Our approach needs to precisely cover almost all (if not all) users' sensitive information, restricted by the sensitive permissions of our interest. This is non-trivial due to the incomplete Android documentation and diverse permission enforcement mechanisms in Android.

- **High accuracy and scalability.** Our approach should minimize the inaccuracy incurred by possible under- and over-approximation during our implementation. Besides, our technique must be efficient for analyzing real-world apps at a large scale.

3. APPROACH OVERVIEW

In the section, we describe the rationale behind our justification approach and its workflow, and then present an example to illustrate how our approach justifies the sensitive information transmission through a real-world app.

3.1 Design Rationale

The key to solve the justification problem is to identify if an app's sensitive information transmission could be used to fulfill some app function. To start, we study *how an app provides functions to mobile phone users*. We realize that the functions of a mobile app in smartphones are experienced by users during their interactions with the app. During the interactions, users are prompted by the changes of sensible phone states (SPS) (e.g., display, sound, vibration and light). Here sensible phones states are defined as phone output events that can be directly sensed by phone users. *In other words, app functions are provided to users via* SPS. Without leading to any SPS *directly or indirectly*, the function of an app is not meaningful to phone users as it cannot be experienced by users. Hence, if a sensitive information transmission cannot cause the change of any SPS, we consider it unnecessary and hence unjustifiable. Otherwise, we consider it justifiable. Note that in the PC world, similar rationale has been adopted by Privacy Oracle [17] and TightLip [33] to detect sensitive information leakage by third-party apps. Their ideas are to apply black-box based differential testing to identify the existence of sensitive user inputs in outbound network traffic by mapping the discrepancy in output network traffic to different inputs.

Figure 1 shows our overall workflow, which answers *how an app's sensitive information transmission is used to provide functions to users* by linking users' sensitive information (SI) with app functions in terms of SPS. In the figure, SI is first read (often further transformed) and delivered to a remote server for computing or other purposes. This is an outbound information flow. Then, if a response from the server is received by the app and ultimately used to change some SPS *directly or indirectly*, we call this inbound information flow *sensible information reception (SIR)*. If an inbound information flow is not a SIR, it will not be sensed by the phone user, so we will not use it to justify any SI transmission. Note that without this rule, an attacker may easily introduces random inbound information flows to justify illegal SI transmissions. We will discuss this problem again in our Security Analysis section. Finally, once we have all the information flows of interest, we want to link inbound and outbound information flows. If a SI transmission cannot be linked to any SIR, it is unjustifiable; otherwise, it is justifiable.

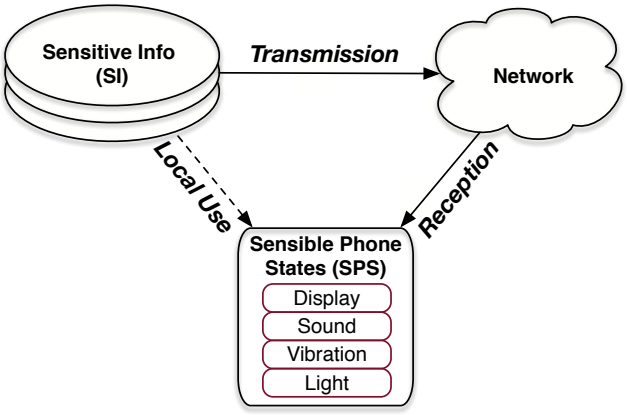

Figure 1: Design workflow: linking sensitive information with app functions

From Figure 1, we can see that the app's sensitive information may also be consumed *locally* to change some SPS. Such local flow information could be useful for other analysis purpose, but for this work, we do not study it. Last, from our formalization, we can see a limitation of our approach: it cannot justify sensitive information flows in some service apps, which run in the background and have no user interface at all.

3.2 An Motivating Example

We present a motivating example to elaborate our proposed analysis approach, as shown in Figure 2.

`com.inspireadart.niceweather` is a popular weather forecast app in Google Play app store. We start from analyzing the app's SI transmission. In a discovered SI flow, the phone location information is first read, transformed to the locality information and returned from the `getLocation` method. Then, the locality information is passed to the `getForecastWeatherData` method in a background task (`doInBackground`). Further, the locality information is put into a URL string, and the URL is finally used to open a HTTP connection after a few manipulations in the `getForecastWeatherData` method.

Next, we analyze the app's sensible information reception SIR. In the discovered SIR flow, the received information is first read from a HTTP connection and returned after a few manipulations from the `getForecastWeatherData` method. Then, the returned information is passed to the `getForecastWeather` method for parsing and the method returns a `WeatherForecast` object in a background task (`doInBackground`). Further, the background task returns the `WeatherForecast` object, which is in turn passed to the `onPostExecute` method as its parameter. In the `onPostExecute` method, the `WeatherForecast` object is passed to the `updateScreen` method. Finally, the extracted information from the `WeatherForecast` object flows into the framework API `setText` to change the text of a `TextView` in the `updateScreen` method. We note that here we show only one of the discovered information flows from the `WeatherForecast` object to SPS due to space limit. In reality, the ex-

Figure 2: A motivating example with an Android app

tracted information from the `WeatherForecast` object also flows into several other framework APIs such as `setImageDrawable` and `setBackgroundResource` to change SPS. We also note that the intermediate representation in each message box in the figure shows an information flow in a method; hence, adjacent lines in a message box may not be adjacent in actual bytecode.

Last but not least, we can see that the discovered SIR is correlated to the discovered SI transmission because they use the exact same network connection (`<HttpURLConnection: void connect()>`). Therefore, through our two-stage information flow analysis, we conclude that this SI transmission is used to fulfill app's functions, and hence *justifiable*.

4. DROIDJUST: OVERVIEW AND SYSTEM DESIGN

This section starts with an overview of the DroidJust's system design, and then describes its details.

4.1 Overview

Figure 3 depicts the overall architecture of DroidJust to justify an app's SI transmission. It takes the following major steps.

1. **Preprocessing.** An Android apk file consists of a Dalvik executable file, manifest files, native libraries, and resources. In this step, DroidJust decomposes the apk file and transforms the Dalvik bytecode executable file into the Jimple representation, which is a typed-3 address intermediate representation suitable for analysis and optimization on the Soot framework.
2. **Sensitive information transmission analysis.** In this step, DroidJust searches the app for SI flows by parsing the permission specifications from PScout [6] and the outgoing channels where the SI flows can reach, and using static taint analysis to identify the SI transmission from the SI (as sources) to the outgoing channels (as sinks).
3. **Sensible information reception analysis.** In the step, DroidJust searches the app for inbound network flows and the framework APIs that can change SPS. This is done by parsing the Android documentation, and using static taint analysis to identify the SIR, with inbound network flows as sources and the framework APIs that can change SPS as sinks.
4. **Correlation and justification.** After the SI transmission and SIR analysis, DroidJust correlates the identified transmissions and reception flows in an attempt to justify all the SI transmissions, and finally determines if a SI transmission is justifiable.

4.2 Sensitive Information Transmission Analysis

In the subsection, we define the users' SI and show how DroidJust identifies them as taint sources. In addition, we show how to identify the outgoing channels as taint sinks. The actual static taint analysis process will be explained in Section 4.4.

4.2.1 Sources

Sensitive information. There are many kinds of SI in Android apps, and currently DroidJust covers ten types of

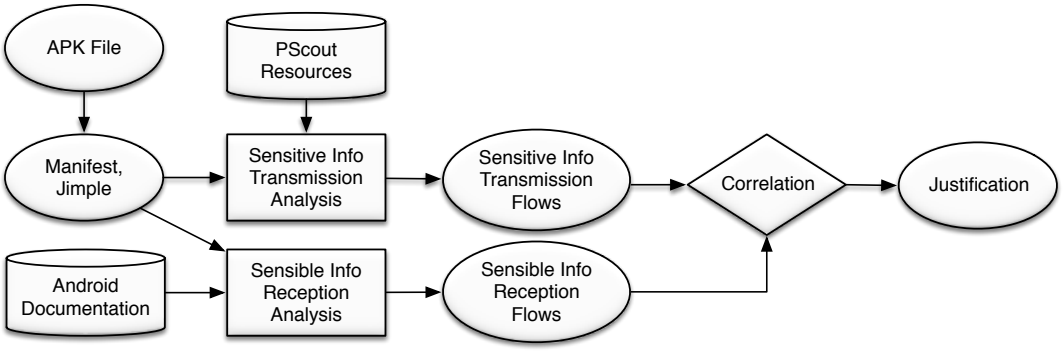

Figure 3: Overall Architecture of DroidJust

SI: phone information (such as device ID and phone number), contacts, messages, user profile, location information, social stream data, calendar information, user accounts, call logs, and browsing history and bookmarks. Android uses security permissions to restrict apps to access *SI*. Particularly, there are 12 Android permissions corresponding to the ten types of *SI*. Except for messages and location information, each type of the aforementioned *SI* is protected by one permission for *read* access. For example, `READ_PHONE_STATE` corresponds to phone information and `READ_CONTACTS` corresponds to contacts. For messages and location information, there are two permissions for each. `READ_SMS` and `RECEIVE_SMS` grant *read* access to SMS messages while `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` grant *read* access to phone's location information.

Label actual sensitive data. Researchers have proposed many tools to identify sensitive data based on Android permissions [11, 12, 15, 18, 20, 21, 31]. However, it is still a big challenge to discover all possible sensitive data sources due to either the incomplete Android documentation or the diverse permission enforcement mechanisms in Android. For better coverage, DroidJust utilizes the permission specification of PScout [6] to identify most (if not all) *SI* sources related to the above permissions. PScout is known as a tool that extracts a relatively complete permission specification from Android. In particular, there are three types of permission-related resources in PScout. The first type (T1) is documented and undocumented framework APIs that retrieve *SI* through returns or callbacks. The second type (T2) is privileged intent actions, which are associated with `IntentFilter` and `BroadcastReceiver` classes to request *SI*. The third type (T3) is URI fields and strings that are identifiers of content providers that manage *SI* in Android.

For T1 (i.e., framework APIs), a major challenge is to map it to actual sensitive data, given that not all return types are used to store actual sensitive data and the callbacks to retrieve sensitive data are very diverse. To overcome the challenge, we handle those framework APIs in the following manner. First, we filter out the non-return APIs and then, based on the Android source code, we manually check the remaining APIs to identify the return types that can be used to store actual sensitive data. As a result, we get a list of 39 return types (of which 21 return types are unique) for the 12 permissions, and we find that 575 framework APIs directly return actual sensitive data for the 12 permissions in Android 4.1.1.

Further, to identify all the possible callbacks in those APIs for retrieving actual sensitive data, we need to get an exhaustive list of the APIs that can retrieve the actual sensitive data through callbacks, as well as the mapping from the APIs to the callback classes, methods (i.e., handlers) and parameters. In practice, we use an automatic filtering method to identify the APIs' parameters that belong to or inherit from the following types: `Receiver`, `Listener`, `Callback`, `PendingIntent` and `Binder`, and then manually check their definitions to find the right callback methods and parameters based on the Android source code. Finally, we obtain a list of 254 framework APIs. By utilizing the refined framework APIs (575+254) and additional mapping information, DroidJust can label the actual sensitive data invoked by the framework APIs in PScout resources more accurately and completely.

Registering a `BroadcastReceiver` with an `IntentFilter` is another way to retrieve *SI* in Android. Hence, the second type of PScout resources (T2) is privileged intent actions that can be added by an `IntentFilter` to gain access to the corresponding *SI*. For example, `"android.provider.Telephony.SMS_RECEIVED"` is a privileged intent action to receive all incoming SMS messages, and a `BroadcastReceiver` class can acquire the incoming messages by registration with an `IntentFilter` including the intent action. Particularly, an intent containing the incoming message is passed to the `onReceive` method of the registered `BroadcastReceiver`.

In Android, there are two ways to register a `BroadcastReceiver` with an `IntentFilter`. One way is to register in manifest files statically. In this case, DroidJust parses manifest files to identify the `BroadcastReceiver` classes that are registered with an `IntentFilter`, and then label the `Intent` parameters of their `onReceive` methods as the actual sensitive data. The second way is to register a `BroadcastReceiver` dynamically. Particularly, the app code can call the method `registerReceiver` at runtime to register a `BroadcastReceiver` with an `IntentFilter`. In this case, DroidJust searches the app for the strings that are equal to the privileged intent actions, then performs *static taint analysis* from the strings (as sources) to `registerReceiver` methods (as sinks) to identify the `BroadcastReceiver` classes that can receive the *SI*, and finally label the `Intent` parameters of their and their subclasses' `onReceive` methods as actual sensitive data.

The third type of resources (T3) in PScout is related

to content providers. In Android, content providers also manage access to certain SI. To retrieve the referenced SI, app developers can use a `ContentResolver` object to resolve a content `Uri` object by calling its query method. Specifically, there are two ways to obtain a content `Uri` object in Android. One way is to directly construct it by encoding a string and the other way is to directly fetch a content `Uri` object from the field of a framework class. For example, constructing a `Uri` object by encoding the string `"content://com.android.contacts"` gives the exact same `Uri` object as `android.provider.ContactsContract.AUTHORITY_URI` gives. Hence, T3 is those strings and fields that can be used to construct or directly fetch `Uri` objects to retrieve SI. DroidJust searches an app for the `Uri` objects that are constructed by the strings or directly fetched from the fields, then performs *static taint analysis* from the found `Uri` objects (as sources) to the `query` method of the `ContentResolver` object (as sinks). It finally labels the result of the query (a `Cursor` type) as actual sensitive data.

4.2.2 Sinks

The retrieved SI can directly flow to the outer world through several channels. Below we describe two most common channels, which are currently covered in DroidJust.

Internet. Android apps can access the Internet and deliver SI in several ways. A common way is to employ a socket-like API or a high-level HTTP client to send out SI. We collect all such APIs from `java.net`, `javax.net` and `org.apache.http` packages. Besides, Android apps may embed SI into a URL and use the Android webkit APIs such as `<WebView: void loadUrl(String URL)>` to deliver SI to the network. Hence, we also collect the related framework APIs from the `android.webkit` package as potential sinks.

SMS. SMS is another popular channel to deliver users' SI, especially for malware. App developers can use the framework APIs in `SmsManager` package to send a message. Hence, we collect a list of the framework APIs from the `SmsManager` package as sinks.

4.3 Sensible Information Reception Analysis

Next, we identify both inbound information flows and sensible phone states (SPS) by parsing the Android documentation. We will delay the description on static taint analysis from the inbound information (as sources) to the SPS (as sinks) in Section 4.4.

4.3.1 Sources

Corresponding to the two types of sinks for outbound SI transmissions, we also consider inbound information flows from these two channels: the Internet and SMS.

Internet. Android apps can receive information from the network by employing a socket-like API or a high-level HTTP client. We collect a list of the related framework APIs from the `java.net`, `javax.net` and `org.apache.http` packages to identify the sources. Besides, Android apps can receive network data by calling the Android webkit APIs. We collect the related framework APIs from the `android.webkit` package to identify the sources.

SMS. We consider the incoming text messages as another source of SIR in our work. Android apps receive incoming text messages by registering a `BroardcastReceiver` with the intent action `android.provider.Telephony.SMS_RECEIVED`. To cover the source, we label the corresponding `onReceive` methods and identify the incoming `Intent` parameters as sources (as shown in 4.2.1).

4.3.2 Sinks

Android apps use framework APIs to change the SPS. For example, `<TextView: void setText(CharSequence)>` is a framework API to change the display of a text editor widget; `<Vibrator: void vibrate(long)>` is used to cause the phone to vibrate. We collect the framework APIs that can change the SPS in four different ways, including display, sound, vibration, and light, by parsing the Android 4.1.1 documentation. In general, many framework APIs can change the SPS via display. Our selection strategy for this type of APIs is to first label all the subclasses of `android.view.View`, because this class represents the most basic building block for UI components in Android. We then manually identify the methods that can change SPS by checking their functions in the Android documentation. Based on our observation, most APIs that can change SPS have a prefix of "set" in their method names. For sound, vibration and light, they have much less framework APIs than the display-related APIs. Hence, we manually find the related classes to collect their methods that can change SPS. Finally, we collect totally 249 Android framework APIs that can change SPS. Table 1 gives a summary of our collected Android framework APIs that are able to change SPS.

Type	Method Name	Quantity
display	setText, setTitle, setIcon, etc.	232
sound	setDataSource, setSound, etc.	11
vibration	setVibrate, vibrate	4
light	setLights	2

Table 1: Android framework APIs that are able to change sensible phone states

4.4 Static Taint Analysis

To identify the data flows from different kinds of sources to different kinds of sinks, DroidJust uses static taint analysis intensively. Specifically, we have the following static taint analysis tasks: 1) from an intent action string to a `registerReceiver` method (in Section 4.2.1), 2) from a `Uri` object to a `query` method (in Section 4.2.1), 3) from the actual sensitive data to the outgoing channels (in Section 4.2), 4) from the inbound information to the SPS (in Section 4.3), and 5) from a URL string to network socket or a high-level HTTP client (in Section 4.5).

DroidJust models the static taint analysis problem within the IFDS [27] framework for inter-procedural distributive subset problems. In practice, DroidJust extends Soot [29], Heros [7] and FlowDroid [5] to provide inter-procedural data-flow analysis. Particularly, FlowDroid generates a dummy main method based on a precise modeling of Android lifecycle and flow functions, which define an IFDS analysis problem; Soot generates a call graph and an inter-procedural control-flow graph (ICFG) from the dummy main method; Heros provides template-driven inter-procedural data-flow analysis by taking as the input flow functions and the ICFG; and DroidJust identifies different kinds of sources and sinks for the inter-procedural data-flow analysis and supports additional indirect static taint analysis (as described below).

Additional Indirect Static Taint Analysis. In practice, we find that the state-of-the-art static taint analysis is ineffective to discover a significant amount of data flows, particularly in the aforementioned tasks 3) and 4), due to the heavy use of data medium in Android apps. That is, tainted data could be first stored into a data medium and later delivered to a sink through data medium. This is very common in Android development since app developers prefer to use data media (e.g., SQLite) as the backend of displayed content. To handle this challenge, DroidJust performs additional *indirect* data flow analysis at two stages: first from sources to the data media, and then from the tainted data media to sinks. In general, there are four types of data media in Android: SharedPreference, ContentProvider, SQLite database and File. Each type of data medium has its own *unique identifier*, and DroidJust taints data media at two stages according to the *unique identifier*. Specifically, SharedPreference uses both `Context` and a filename (a string) to uniquely identify a preference file; ContentProvider uses `Uri` to uniquely identify a data repository; SQLite uses a table name (a string) to identify a table on a default database; and File uses a filename (a string) to identify a stored file. By launching the two-stage static taint analysis, DroidJust is able to discover almost all data flows.

4.5 Correlation and Justification

After identifying the app's SI transmissions and SIR, DroidJust tries to justify each of the SI transmission flows by linking it to an SIR flow. This correlation task is not easy since DroidJust cannot acquire and analyze the server-side logic. To try the best, DroidJust solves it in the following manner. SI transmission flows deliver the sensitive information either via the Internet or SMS. Considering a SI transmission flow delivering the sensitive information via the Internet, if the transmission flows into the Android webkit APIs, it is justifiable since the transmission displays a `WebView` to users in phone screen. Otherwise, it means the transmission flows into a socket-like API or a high-level HTTP client.

In the latter case, DroidJust first finds if the transmission is *synchronous* to any SPS flow. Specifically, DroidJust checks if the SI transmission flow and the SIR flow share the same network socket or HTTP client. If true, they are correlated. Otherwise, DroidJust continues to check if a SI transmission flow is *asynchronous* to any SIR flow. More specifically, DroidJust checks if the destination of the SI transmission and the source of the SIR are the same. In other words, DroidJust checks if the network server that delivers information to the SPS is the same server where the SI flow goes to.

There are two tasks to map the server names. The first task is to extract the network addresses from each of the inbound and outbound information flow. DroidJust first identifies all the URL- or IP-like strings and then uses *static taint analysis* to find the strings that flow into the network connection of the SI transmission or SIR. The identified strings are the network addresses of the SI transmission or reception. The second task is to check if these network addresses refer to the same network server. The simplest way is to compare the hostnames of the network addresses. However, in reality, an app may use different hostnames for the same server. To cope with this situation, we further check whether the IP addresses of the hostnames are equal. Finally, DroidJust justifies the SI transmission if either the hostnames or the IP addresses of the hostnames are the same. In its implementation, DroidJust uses the standard Java API (`<URI: String getHost()>`) to extract hostnames from URL- or IP-like strings and nslookup to resolve hostnames to IP addresses.

On the other hand, considering a SI transmission is through SMS, DroidJust simply checks if the app receives any incoming messages to change SPS. If yes, the transmission is justifiable; otherwise, the transmission is unjustifiable.

5. EXPERIMENTAL EVALUATION

To evaluate the effectiveness, accuracy and efficiency of DroidJust, we perform experiments on 6111 Google Play apps and 340 known SI-stealing Android malware collected from VirusTotal [3]. Next we report the detailed results and our findings.

5.1 Evaluation on Google Play Apps

In the experiment, we evaluate DroidJust over 6111 apps, randomly downloaded from the Google play store during March, 2014. Based on a report from Andrubis [30], only 1.6% Google Play apps are identified as malware by antivirus vendors. Hence, we expect most of these downloaded apps to be benign and not leak user privacy. By scanning these apps with DroidJust, we evaluate whether DroidJust can precisely identify benign apps, particularly those delivering SI to the network.

We setup our evaluation on a cluster with hundreds of Intel Xeon E5-2665 2.40 GHz processors (16 cores per processor). Each analysis task (for analyzing an app) is assigned to a cluster node with 4 cores and 16 GB physical memory, which runs JDK 1.7.0_21. DroidJust takes about 85 hours of CPU time to analyze 6111 Google play apps and 12 hours of CPU time to analyze 340 known malware. On average, each Google play app takes about 50 seconds of CPU time and each known malware about 128 seconds of CPU time. Hence, DroidJust is definitely an affordable tool for antivirus vendors or Android market operators.

During the evaluation, we notice that DroidJust cannot analyze some apps due to either insufficient memory or failure of type resolving. Basically, DroidJust shares the same problem with other FlowDroid-dependent tools [19]. We start by analyzing 6111 apps, among which 1092 apps failed to go through. Thus, below we show our experimental results over the remaining 5019 apps.

Results. Figure 4 illustrates the analysis results. Among 5019 Google play apps, 95.82% apps do not send out users' SI, while 4.18% (210) apps transmits users' SI via Internet or SMS. Among those transmitting SI, 3.61% (181) apps' SI transmissions are justifiable while 0.58% (29) apps' SI transmissions are unjustifiable.

Validation. We manually check these 29 apps by analyzing their intermediate representation (Jimple) and read their descriptions in Google Play to understand their functions and validate our detection results. After validation, we classify these 29 apps into the following categories.

(C1) Stealthily send out SI for app functions. There are ten apps that stealthily send out users' SI for application functions including anti-theft, location-tracking and spying. Those apps have clear descriptions about their stealthy behavior in Google Play. For anti-theft apps (e.g., `avsolu-`

Package Name	Leaked Sensitive Info	Dynamic Tainting [1]	Still on Google Play? [2]	VirusTotal Score [2]
com.controlaltkill.autoball	Phone number	No	Yes	0/57
com.jb.gosms.pctheme.loving_bears	IMEI	Yes	Yes	0/57
com.kingdom_card_wcm777_1	Location	No	Yes	0/57
com.kokovoin.homedesign	IMEI	No	Yes	1/56
com.mapnavigation	IMEI	Yes	Yes	12/57
com.necta.aircall_accept.free	IMEI	Yes	Yes	7/57
com.pixoplay.candyshooter	IMEI	Yes	Yes	0/57
com.topdisk.launcher	IMEI	N/A [3]	Yes	0/57
cz.prilozany.android.compass	IMEI, Location	Partial [4]	Yes	0/57
fr.pb.tvmobile	IMEI	No	No	0/56
lk.bhasha.sett.hindi	IMEI	Yes	Yes	3/57
lk.bhasha.vishwa	IMEI	No	No	0/57
me.chatcast.kaomoji	Gmail account name	No	Yes	0/57
me.zed_0xff.android.alchemy	IMEI	Yes	Yes	2/57
mx.websec.mac2wepkey.hhg5xx	Location	No	Yes	0/57

Table 2: (C3) Identified Google play apps that send out users' sensitive information not for functions. Notes: 1. we use Andrubis for dynamic taint analysis, whose dynamic taint analysis is based on TaintDroid; 2. we update till Feb. 13, 2015; 3. the file exceeds the maximum size limit (8MB) restricted by Andrubis; 4. only the IMEI leak is identified.

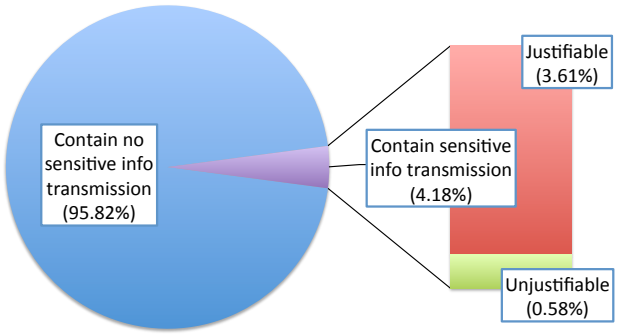

Figure 4: Analysis results for Google Play apps

`tion.version1`), once a thief changes the SIM card of a stolen phone, they immediately notify the original users of phone information (e.g., phone number, IMEI, and location) of the new SIM card via SMS in a background task. This communication is designed to be one-way. Location-tracking apps (e.g., `gaugler.backitude`) send out locations via Internet or SMS to track a mobile device in real-time. Besides, there are spyware whose main function is to spy on users, as stated in their descriptions. For example, `com.dona.messagespoofing` is an intentionally designed spyware to stealthily forward all incoming SMS messages to a designated phone number after the first-time setup. In summary, those apps are supposed to stealthily send out SI without users' awareness. In terms of behavior, they are very similar to spyware that steal users' sensitive information. Hence, it is indeed appropriate for DroidJust to label them as unjustifiable from the perspective of code behavior. We note that privacy analysts can easily distinguish all these apps from malicious spyware by reading their descriptions.

(C2) Analytics libraries. Four apps containing analytics libraries are found sending out users' phone information to remote servers. We recognize these analytics libraries by checking the hostnames of their network servers. They have affiliations with the mobile application solution providers including accelerator, crittercism and kontagent. DroidJust does not justify the SI transmissions by these analytics libraries because they do not provide any function back to users. We note that, in practice, an analytics library is often bundled with the same provider's advertisement library and therefore our tool will justify the sensitive information transmission in the analytics library by identifying the SIR in the advertisement library.

(C3) Stealthily send out SI but not for app functions. We identify 15 apps that stealthily send out users' SI to remote servers, but such SI transmissions cannot be justified. Those apps do not describe anything about their stealthy behavior in their descriptions in Google Play. To avoid possible false alarms, we manually and carefully check their app logic by analyzing their intermediate representation (Jimple) to ensure that the discovered SI transmission does not provide any function to users. Besides, we use the dynamic taint analysis tool Andrubis to expose their privacy leak behavior in runtime.

Table 2 shows a summarized result for these 15 apps. From left to right, the table shows app's package name, leaked SI identified by DroidJust, if Andrubis can identify the same leak, if the app is still available for download on Google Play, and how many antivirus engines in VirusTotal identify the app as malicious. From this table, our observations and findings are as follows. First, we can see that the most frequently leaked SI is device ID (IMEI). This is expected based on the past research work [9, 35]. Second, Andrubis's dynamic taint analysis did not identify all the SI transmissions in 8 apps, mostly due to the failure in generating appropriate inputs. For example, `com.kingdom_card_wcm777_1` and `lk.bhasha.vishwa` require users to provide correct authentication information for a mobile payment account and the Facebook account, respectively, at the very beginning, and dynamic taint analysis fails to bypass the authentication in runtime. Third, after almost one year, Google Play only removed two apps while the remaining 13 apps are still available for download. Last

but not least, 10 of these 15 apps cannot be detected by any antivirus engines in VirusTotal.

5.2 Evaluation on Known Malware

In this experiment, we evaluate DroidJust on 340 malware known for stealing users' SI. To collect them, we first collect a list of malware families that are known for stealing user private information from Genome [35] and Forensics blog [2], and then download the apps related to these malware families from VirusTotal by using their advanced reverse search system [3]. We run DroidJust against these 340 apps to evaluate its detection precision. We start with 340 apps and unfortunately 42 apps fail to go through due to the same reason as we mentioned above. Thus, here we only show our experimental results for the rest 298 apps.

Results. Table 3 shows the analysis results for each malware family. There are 43 malware families in total. For each malware family, the number of samples is between 1 and 31. In the table, the column *Positive* gives the number of samples that are identified to contain unjustifiable SI transmission and column *Negative* gives the number of samples that are identified to contain only justifiable SI transmissions. The total number of positive outcomes is 274, while the total number of negative outcomes is 24. Thus, the detection rate is 91.94%. The malware samples leak five kinds of SI, as shown in the middle of the table: phone information (P), contacts (C), messages (S), locations (L) and accounts (A). We note that the marks in each row indicate the union of sensitive information types that are leaked by all samples in the malware family; as such, not every sample in a malware family leaks all marked types. We can see that phone information is the most leaked SI among the 43 malware families.

Validation. We manually inspect the 24 apps with negative outcomes by analyzing their intermediate representation. There are two main sources of false negatives for DroidJust. One is because of dynamic code loading. That is, some malicious apps download and install other malware after exploiting certain root access vulnerability. It is the dynamically downloaded malware code that leaks user privacy. Static taint analysis inherently cannot detect the leakage behavior by the malware installed later. The second reason is due to the inaccurate callback-based lifecycle modeling in FlowDroid [14].

Malware familiy	Sensitive information					Pos.	Neg.
	P	S	L	C	A		
anserver	x					30	1
avpass	x					1	0
backflash	x	x				2	0
basebridge	x	x				27	1
beanbot	x	x				6	0
bgserv	x	x				3	1
droiddreamlight	x	x		x	x	25	2
droidkungfu	x					20	2
extension	x					1	0
fakeangry	x					5	0
fakebank	x			x		8	0
fakemart		x				1	0
faketaobao		x				2	1
fjcon	x	x	x			4	0
fokonge	x					19	1
geinimi	x	x	x			17	1
ggtracker	x	x				6	1
gingermaster	x		x			16	3
godwon				x	x	3	1
golddream	x	x		x	x	29	0
hongtoutou	x	x				17	1
kmin	x	x				29	0
lena	x					2	0
loozfon	x			x		3	0
mobilespy	x	x		x		4	1
mobiletx	x					2	1
pjapps	x					22	1
plankton	x					12	1
roguelemon	x					1	0
roidsec		x	x	x		2	0
sinpon		x	x	x		2	0
skullkey	x					1	0
smspacem		x		x		1	0
sndapps	x				x	9	0
spitmo	x	x				3	1
spyoo	x					3	0
ssucl	x	x		x	x	1	0
tetus	x					1	0
typstu	x				x	9	0
usbcleaver	x					1	0
vdloader	x					1	0
yzhc	x					11	2
zitmo	x	x				2	1
Total (43)	37	20	5	10	6	274	24

Table 3: Malware families featuring privacy leakage
P: Phone information; C: Contacts; S: Messages;
L: Locations; A: Accounts.

6. DISCUSSION

In this section, we discuss the limitations of our approach.

Security Analysis. In our design, we ignore the *meaningless* inbound flows (i.e., those not leading to any SPS) to prevent attackers from introducing noisy inbound flows to evade our detection. Determined attackers, however, may manage to introduce noisy inbound flows that indeed lead to some SPS. Since SPS is sensible to users, the attacker will need to ensure that such noisy flows will only cause minimal changes of phone states to not degrade the usability and functionality of the app. We will examine the practicality of this and other attacks and accordingly design possible countermeasures in our future work.

Implicit Information Flows. Sensitive information can propagate in other channels than direct channels, such as control flow and timing channels. It is very challenging to detect and track these channels. In this work, we do not consider tracking implicit information flows. The limitation is also shared by other taint analysis tools, such as TaintDroid [9] and PiOS [8]. We leave it as our future work to support the discovery of implicit information flows.

Java Reflection & Native Code. Static information flow analysis always has the trouble to handle Java reflection and native code due to the lack of full knowledge on Java reflective calls and JNI calls [11]. In our work, we use taint wrappers with various crafted function summaries to partially resolve the propagation through Java Reflection,

which however may introduce some false positives. We do not deal with native code for data propagation. Potentially, we could model the well-known JNI calls and thereby create the corresponding taint wrappers for the calls to exercise static data propagation. We leave it as our future work to enhance our tool.

7. RELATED WORK

Most prior approaches to detecting privacy leakage in mobile apps use either static or dynamic analysis. TaintDroid [9] is a dynamic analysis tool for monitoring potential privacy leakage in Android apps by modifying Dalvik virtual machine and dynamically instrumenting Dalvik bytecode instructions. PiOS *et al.* [8] is a static analysis tool for discovering possible leaks of SI from a mobile device to third parties in iOS devices. Enck *et al.* [10] use ded [22], a re-targeting tool, to convert a Dalvik executable back to Java source code, and leverage a commercial Java source code static analysis tool named Fortify 360 [1] to detect suspicious information flow. AndroidLeaks [12] is another static analysis tool to detect potential privacy leakage in Android applications by leveraging the WALA [4] framework. Mann *et al.* [21] also proposed a static taint analysis based framework by using their self-crafted abstract Dalvik virtual machine instruction set and a security type system. FlowDroid [5] is a precise context, flow, field, object-sensitive and lifecycle-aware static taint analysis tool to detect SI transmissions in Android apps. FlowDroid uses SuSi [25], a machine-learning approach to identifying an app's sensitive information sources and sinks. To summarize, all these approaches are capable of detecting an app's SI transmissions, but they are not designed to justify the SI transmission automatically.

Several approaches have focused on the justification of an app's SI transmission by examining the contextual information of the leakage. AppIntent [32] is an analysis tool to provide a human analyst with the contextual information of privacy data transmission, particularly, the chain of events leading to the triggering of a transmission, to help justify discovered SI transmissions. However, the approach still needs human effort to justify every discovered SI flow. Tripp *et al.* proposed a bayesian approach to statistically classify SI transmissions as legitimate or illegitimate based on the evidences arising at the release point [28]. The effectiveness of the approach highly depends on the select feature of the evidences for their statistical inference, which is the similarity between actual sensitive data and the data about to be released. Different to those two approaches, which consider the evidences arising before and at the release point, our approach uses the evidence arising after the release point for privacy leakage detection. Zhang *et al.* proposed Capper, a bytecode rewriting tool, to instrument Android apps to alert users on SI transmissions in runtime and enable users to allow/deny the transmission [34] . Market providers and antivirus vendors, however, cannot use the reactive approach to perform large-scale detection.

Past research work have also demonstrated the strong relationship between an app's meta information and its declared permissions. Pandita *et al.* and Qu *et al.* proposed WHYPER and AutoCog to automatically infer an app's necessary permissions from its description by using natural language processing [23, 24]. These approaches can be potentially used to provide additional useful information to justify an app's SI transmission. However, it is nearly impossible to only use meta information to justify an app's SI transmission because meta information is often very high-level, incomplete and sometimes inaccurate in reflecting all permission needs. Note that DroidJust is not designed for permission analysis, but rather a tool for sensitive information flow analysis. Because a privacy-sensitive permission might be needed in multiple sensitive information flows in an app, even if the purpose of a permission is justified, a dependent individual SI flow may still be unjustifiable. For example, a malicious weather forecast app may be justified for the location permission based on its description, but one of its SI flows stealthily sending to an unknown third party cannot be justified. This indicates that these tools and DroidJust work at different granularities and may complement each other.

8. CONCLUSION

We present DroidJust, an automated approach to justifying an app's *SI* transmission by bridging the gap between *SI* transmission and app's functionality. It uses static taint analysis to first discover the *SI* transmissions to the network, then discover the information receptions (from the network) that serve application functions, and finally justify the discovered *SI* information transmissions by correlating the outbound and inbound information flows. Our evaluation on real-world Android apps and known malware demonstrates that DroidJust can effectively and efficiently analyze both benign apps and malware. x‘

9. ACKNOWLEDGMENTS

We thank the reviewers for the valuable comments. This work was supported in part by NSF grant CCF-1320605. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. Government. We gratefully acknowledge the support of VirusTotal for this work and the Research Computing and Cyberinfrastructure Unit of Information Technology Services at The Pennsylvania State University for providing advanced computing resources and services that have contributed to the research results reported in this work.

10. REFERENCES

[1] Fortify 360 source code analyzer. `http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812#.U3U1YlhdXKo`.

[2] Mobile-Sandbox. `http://forensics.spreitzenbarth.de/android-malware/`.

[3] VirusTotal. `https://www.virustotal.com/`.

[4] WALA, T. J. Watson libraries for analysis. `http://wala.sourceforge.net/`.

[5] S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI' 14*, 2014.

[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission

specification. In *Proceedings of the 2012 ACM conference on Computer and communications security, CCS' 12*, pages 217–228. ACM, 2012.

[7] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 3–8. ACM, 2012.

[8] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS' 11*, 2011.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI' 10*, volume 10, pages 1–6, 2010.

[10] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security Symposium*, 2011.

[11] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security, CCS' 11*, pages 627–638. ACM, 2011.

[12] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.

[13] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 21–26. ACM, 2011.

[14] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of android applications in droidsafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS'15*, 2015.

[15] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851. ACM, 2013.

[16] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security, CCS' 11*, pages 639–652. ACM, 2011.

[17] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. 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, CCS' 08*, pages 279–288. ACM, 2008.

[18] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 2012.

[19] L. Li, A. Bartel, J. Klein, and Y. L. Traon. Automatically exploiting potential component leaks in android applications. In *Trust, Security and Privacy in Computing and Communications, TrustCom' 14*, pages 388–397. IEEE, 2014.

[20] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security, CCS' 12*, pages 229–240. ACM, 2012.

[21] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[22] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, 2012.

[23] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security*, volume 13, 2013.

[24] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS' 14*, pages 1354–1365. ACM, 2014.

[25] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Network and Distributed System Security Symposium, NDSS' 14*, 2014.

[26] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

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

[28] O. Tripp and J. Rubin. A bayesian approach to privacy enforcement in smartphones. In *USENIX Security*, 2014.

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[30] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001*, 2014.

[31] Z. Yang and M. Yang. Leakminer: Detect information leakage on android with static taint analysis. In *WCSE' 12*, pages 101–104. IEEE, 2012.

[32] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, CCS' 13*, pages 1043–1054. ACM, 2013.

[33] A. R. Yumerefendi, B. Mickle, and L. P. Cox.

Tightlip: Keeping applications from spilling the beans. In *NSDI' 07*, 2007.

[34] M. Zhang and H. Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIACCS' 14*, 2014.

[35] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy, S&P' 12*, pages 95–109. IEEE, 2012.