LeakDoctor: Toward Automatically Diagnosing Privacy Leaks in Mobile Applications

XIAOLEI WANG, College of Computer, National University of Defense Technology, China
ANDREA CONTINELLA, Department of Computer Science, University of California Santa Barbara, USA
YUEXIANG YANG, College of Computer, National University of Defense Technology, China
YONGZHONG HE, School of Computer and Information Technology, Beijing Jiaotong University, China
SENCUN ZHU*, Department of Computer Science and Engineering, The Pennsylvania State University, USA

With the enormous popularity of smartphones, millions of mobile apps are developed to provide rich functionalities for users by accessing certain personal data, leading to great privacy concerns. To address this problem, many approaches have been proposed to detect privacy disclosures in mobile apps, but they largely fail to automatically determine whether the privacy disclosures are necessary for the functionality of apps. As a result, security analysts may easily face with a large number of false positives when directly adopting such approaches for app analysis. In this paper, we propose *LeakDoctor*, an analysis system seeking to automatically diagnose privacy leaks by judging if a privacy disclosure from an app is necessary for some functionality of the app. Functionality-irrelevant privacy disclosures are not justifiable, so considered as potential privacy leak cases. To achieve this goal, LeakDoctor integrates *dynamic response differential analysis* with *static response taint analysis*. In addition, it employs a novel technique to locate the program statements of each privacy disclosure. We implement a prototype of LeakDoctor and evaluate it against 1060 apps, which contain 2,095 known disclosure cases. Our experimental results show that LeakDoctor can automatically determine that 71.9% of the privacy disclosure cases indeed serve apps' functionalities and are justifiable. Hence, with the diagnosis results of LeakDoctor, analysts may avoid analyzing many justifiable privacy disclosures and only focus on the those unjustifiable cases.

CCS Concepts: • **Security and privacy → Software security engineering**; **Usability in security and privacy**.

Additional Key Words and Phrases: Privacy Leak, Differential Analysis, Taint Flow Analysis, Mobile Applications

ACM Reference Format:
Xiaolei Wang, Andrea Continella, Yuexiang Yang, Yongzhong He, and Sencun Zhu. 2019. LeakDoctor: Toward Automatically Diagnosing Privacy Leaks in Mobile Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3, 1, Article 28 (March 2019), 25 pages. https://doi.org/10.1145/3314415

*The Corresponding Author.

Authors' addresses: Xiaolei Wang, xiaoleiwang@nudt.edu.cn, College of Computer, National University of Defense Technology, Deya Street 109, Changsha, China, 410000; Andrea Continella, Department of Computer Science, University of California Santa Barbara, Department of Computer Science, 2104 Harold Frank Hall, University of California, Santa Barbara, Santa Barbara, USA, conand@cs.ucsb.edu; Yuexiang Yang, College of Computer, National University of Defense Technology, Changsha, China, yyx@nudt.edu.cn; Yongzhong He, School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China, yzhhe@bjtu.edu.cn; Sencun Zhu, Department of Computer Science and Engineering, The Pennsylvania State University, State College, Pennsylvania, USA, szhu@cse.psu.edu.

2474-9567/2019/3-ART28 $15.00
https://doi.org/10.1145/3314415

1 INTRODUCTION

Today mobile devices, particularly smartphones, have become prevalent all over the world, largely because of their significant computation capacity, convenient communication, and multiple sensing capability, as well as the abundant data they have about users. At the meantime, millions of mobile apps have been developed to provide the rich functionalities for users with these mobile devices. Although smartphones and apps have created many new opportunities and new usage modalities, they have also led to several new challenges. For example, as mobile apps are increasingly used for private and privileged tasks, there are also rising concerns on the consequences of failure to protect or respect user's privacy. Manufacturers and researchers have responded to this concern by introducing various solutions. On one hand, they have proposed a number of privacy related enhancements to the stock OS privacy primitives and extended them in several significant ways, including fine-grained app level permission controls [15], providing obfuscated or anonymized private data to apps [6, 26, 40, 46], giving users privacy recommendations [35], or even looking at new notification modalities to help with user privacy awareness [11]. On the other hand, many approaches have also been proposed to detect privacy disclosures in mobile apps, for example, static control flow and data flow analysis [4, 10, 30] [37, 52, 57], dynamic data flow tracking [14, 43, 53], static-dynamic hybrid method [55], and network traffic analysis [29, 44].

Although many methods have been proposed to detect privacy disclosure in mobile apps, most existing works are coarse-grained, staying on the app level. They can identify what privacy disclosures happen in a mobile app regardless of their necessity and legitimacy, and hence cannot provide accurate analysis results for users and analysts. Firstly, they are unable to help users distinguish whether a privacy-sensitive data item, such as location or their identity, is needed for a core functionality of the app or is just being sent to various third parties. Secondly, as more and more benign apps send out sensitive data for legitimate functions, these approaches cannot justify whether a certain privacy disclosure should happen for app's functionalities, and hence may generate too many false positives. For example, *Google Maps* needs to send out user's location information for driving navigation. A weather app may request user's location to provide weather services. These functionality-necessary false positives may overwhelm security analysts when analyzing the detection results. In our work, a ***privacy disclosure*** is formally defined as a network request sending out one or multiple types of private data, while ***privacy leak*** refers to privacy disclosure which cannot be justified by apps' functionality (i.e., unjustifiable disclosure).

Automatically determining whether a privacy disclosure is necessary for app functionality or not still remains an open problem, which needs more research. The key challenge is related to the fact that automatically diagnosing privacy leak is very difficult without taking into account of the specific purpose and normal functionality of an app, and hence, it is out of reach for most existing analysis tools.

Our Work. In this paper, we focus on analyzing privacy disclosures through network traffic and take a step towards automated privacy leak diagnosis. It is noted that encrypted network traffic is intercepted through a man-in-the-middle proxy based on the mitmproxy library [2]. Our approach mainly leverages the following observations: 1) an app's functionality should be experienced by end users through human sensible phone states (SPS) (e.g., display, sound, etc). 2) for each necessary privacy disclosure in an app, it should directly or indirectly serve some functionality for the app; otherwise, the mobile device may feed the app with fake data without affecting users' experience with the app. In other words, such privacy disclosures are not necessary. Based on these observations, we propose a novel fine-grained approach, called **LeakDoctor**, which aims to automatically detect each privacy disclosure and diagnose its necessity for app functionality. LeakDoctor is designed from both static and dynamic analysis perspectives: 1) Dynamic analysis perspective. LeakDoctor actively changes, at runtime, the values of the leaked private data and observes how such values affect remote responses to determine if they affect app's functionality. 2) Static analysis perspective. LeakDoctor tries to link each privacy disclosure with certain app's functionality (e.g, SPS-related APIs) to determine if it serves app's functionality. A *privacy*

disclosure is ***justifiable*** when it does serve some functionality, or else it is ***unjustifiable*** and hence considered as a *privacy leak*.

To make privacy disclosure analysis more objective and measurable, a design principle of LeakDoctor is to separate techniques from users' privacy preferences by providing diagnosis results based on whether a privacy disclosure provides certain functionality for an app. We understand that some users may differentiate desirable functionality from undesirable ones. For example, privacy advocates may consider it a privacy leak when an app sends out users' locations for advertisement purpose. LeakDoctor is not designed to judge the legitimacy of the above case for users, but tells users whether the disclosure of users' locations has something to do with some app functionality, purely from technical point of view. In other words, the diagnosis result here is purely from the technical aspect, not from the legal aspect. Certainly, depending on the privacy preference of analysts, LeakDoctor can be easily configured to report the diagnosis result (i.e., a privacy leak or not) based on whether and how a privacy disclosure is related to *any* functionality of an app, or related to *some desirable* functionality of the app.

Goals, Scope and Contributions. Generally, LeakDoctor is designed to help developers, app store owners and security analysts understand whether a privacy disclosure does serve some app's functionality or not. ***1)***. We mainly aims to provide objective and measurable diagnosis results of each detected privacy disclosure. Similar to a previous work [11], there are no guarantees on what happens to private data once it leaves the users devices over the network. ***2)***. Our current prototype only focuses on privacy disclosures through network, and it does not handle other communication channels, such as sending requests or receiving responses through SMS. ***3).*** Some apps are designed to stealthily send out user's private data for special functions, including anti-theft, location-tracking, back up (e.g., uploading files to cloud disk), etc. Since such privacy disclosures are often designed to be one-way (i.e., no response), LeakDoctor will label them as privacy leaks. While this sounds like false positive cases, technically speaking LeakDoctor works correctly as intended. It is only a different interpretation of diagnosis results in specific contexts, so it is not a fundamental flaw of LeakDoctor. Security analysts can easily distinguish such cases by reading the apps' descriptions, as explained in detail in Section 7.1. To summarize, we make the following contributions:

- To our best knowledge, we are the first to devise techniques, through static network pair extraction and bytecode instrumentation, to locate the program statements that generate privacy disclosures.
- We develop a new dynamic analysis based technique, namely *response differential analysis*, by actively changing the private data leaked in a privacy disclosure to test whether it affects remote response or not. Aided by this technique, we can further determine whether this privacy disclosure truly affects app's functionality or not.
- Aided by the privacy leak diagnosis of LeakDoctor, we are able to automatically determine whether a specific privacy source is necessary to leak. This capability can support fine-grained access control over private data leak.
- We implement the prototype of LeakDoctor and evaluate it over 1060 apps. Experimental results show that LeakDoctor can perform privacy leak diagnosis with high accuracy and good performance.

2 MOTIVATION AND ASSUMPTIONS

This section discusses a real-world example that motivates our work. As shown in Listing 1, the code snippet first obtains the `Device ID`(line 1) and `phone number`(line 2). The `Device ID` and `phone number` are concatenated (line 3) and encrypted using `AES`(lines 4-8). Then, it sends the encrypted content to a remote server through a `POST` request (lines 11-14). Finally, it receives the response (lines 15-21) and displays it to user through UI (lines 22-25). Based on this code snippet, one can see there are two main research challenges in privacy leak detection of android apps.

The first challenge is how to accurately detect privacy disclosures, especially when various obfuscation techniques are used. To our knowledge, most previous works [4, 14, 48, 49] detect privacy disclosures through either static or dynamic taint data flow tracking. Some of the previous work can also handle privacy disclosures under simple obfuscation, but cannot perform well when more complex obfuscation techniques are used. Recently, a new tool named AGRIGENTO [12] was proposed for obfuscation-resilient privacy disclosure [1] detection through differential analysis of network traffic.

Listing 1. Code Snippet Leaking Device ID and Phone Number

```
1  String device_id=getDeviceId();
2  String phone_number=getLine1Number();
3  String str='id:'+device_id+'number:'+phone_number;
4  byte[] secretKey= "/{_$#%!)".getBytes();
5  SecretKeySpec key = new SecretKeySpec(secretKey,"AES");
6  Cipher cip=Cipher.getInstance("AES/ECB/NoPadding");
   //encrypt str using AES
7  cip.init(ENCRYPT_MODE, key);
8  String encr=new
   String(cip.doFinal(str.getBytes()));
9  String url="http://nma.namcowireless.com/nma/Api
      /Response/protocol/2/store/?";
10 HttpURLConnection conn=(HttpURLConnection)
      new URL(url).openConnection();
   //Instrumentation
11* String log_url=conn.getURL().toString();
12* Log.i('Pair1',log_url);
11 OutputStream os=conn.getOutputStream();
12 BufferedWriter bfWriter = new BufferedWriter
      (OutputStreamWriter(conn.getOutputStream());
13 bfWriter.write(encr);
14 bfWriter.close();
15 InputStreamReader in = new InputStreamReader
      (conn.getInputStream(),"UTF-8");
16 BufferedReader buffer = new BufferedReader(in);
17 String inputData = "";
18 while (true){
19 String inputLine = buffer.readLine();
20 if (inputLine == null) {break;}    //end if
21 InputData = new StringBuilder(inputData).append
      (inputLine).toString();}
22 if(inputData!=""){
23 GameInfo = (TextView)findViewById(R.id.game_info);
24 GameInfo.setText(inputData);}   //end if
25 } //end while
```

Listing 2. Code Snippet from GoldDream Sample

```
1  String Dev_MIEI=getDeviceId();
2  String Dev_SimSSN =getSubscriberId();
3  String Dev_IMSI =getSimSerialNumber();
4  StringBuilder url=new StringBuilder(String.
   valueOf("http://removedDomain"+ "zjRegistUid.
   aspx?")).append("&imei=").append(Dev_MIEI).
   append("&sim=").append(Dev_SimSSN).append
   ("&imsi=").append(Dev_IMSI).toString()
5  HttpURLConnection conn=(HttpURLConnection)
      new URL(url).openConnection();
6  InputStreamReader in = new InputStreamReader
      (conn.getInputStream(),"UTF-8");
7  byte[] b = new byte[in.available()];
8  in.read(b);
9  String str = new String(b);
10 Editor editor = getSharedPreferences("uid",
   2).edit();
11 editor.putString( "uid_v", str);
12 editor.commit(); // store the UID locally
13 ...//get remote tasks using the stored UID
```

The second challenge is how to automatically determine if a privacy disclosure is a privacy leak or not. Although many techniques have been proposed to detect privacy disclosures in mobile apps, they are incapable of automatically judging if a privacy disclosure is a real privacy leak. Simply reporting the existence of privacy disclosures may not be that meaningful to security analysts. To tackle this challenge, a number of solutions have also been proposed, as discussed below.

- **User expectation based method**. For example, AppIntent [57] tries to detect privacy leaks based on users' expectations. However, this approach needs manual work to justify every privacy disclosure and cannot work automatically. Moreover, users' expectations are diverse and most users cannot well understand the contextual information provided by AppIntent. Ferreira et al. [17] present Securacy, an app that aims to

[1]While the paper uses the term privacy leak instead, it is really privacy disclosure in our context.

detect privacy violations by asking users to pre-specify permissions they are concerned with, and alerting users accordingly. Securacy also has a crowd-sourcing module where users can share information about security issues they experienced with an application and provide a rating for other users to consider before first application use. Although Securacy can help protect user's privacy better, it still focuses on permission level and fails to detect unnecessary leaked private data at information flow level.

- **Natural language processing based method**. For example, previous works such as WHYPER [41], AutoCog [42], and [50] leverage nature language processing (NLP) techniques to understand whether the application description of an app reflects its permission usage. While working effectively on the permission level, they cannot be applied to justify privacy disclosure at the data flow level, as shown above.
- **Machine learning based methods**. For example, BayesDroid [49] and LeakSemantic [18] both formulate the privacy leak justification as a machine learning problem based on certain features. However, such methods are probabilistic in nature and their effectiveness highly depends on the selected features and training data sets.
- **Static taint flow analysis based methods**. For example, AAPL [37] was proposed to filter legitimate privacy disclosures through peer voting on detected taint flows. However, AAPL suffers from relatively high false positive rate and false negative rate, because being in the same category does not imply the apps have the same functionality. Hence, its peer voting mechanism may become ineffective. Differently, DroidJust [10] proposes to differentiate privacy leak and disclosure by judging whether a privacy disclosure serves an app's function through static taint analysis. However, DroidJust also suffers from high false positive rate, because it cannot accurately correlate the network response with the corresponding request.

In this work, we mainly focus on solving the second challenge and design an automated and fine-grained approach, **LeakDoctor**, to detect privacy disclosures through network, and further justify whether each detected privacy disclosure is necessary for app functionality or not. To handle the first challenge, we borrow the techniques from AGRIGENTO [12] for privacy disclosure detection, as a front-end component of our overall approach. Note that this is just an implementation choice, because LeakDoctor's diagnosis is generic and it can also adopt other similar techniques, such as MUTAFLOW [38], to detect privacy disclosures (more details in Section 7.3).

3 APPROACH OVERVIEW

The key idea behind our approach to diagnosing privacy leak is to determine whether a detected privacy disclosure affects or serves an app's functionality. If yes, this privacy disclosure is necessary and ***justifiable***; otherwise, it is ***unjustifiable*** and considered as a privacy leak.

To see the point, let us first understand *how an app provides functions to users*. Mobile app's functionality is normally experienced by users during their interactions with the app. During the interactions, users are prompted by the changes of human sensible phone states (SPS) (e.g., display, sound, vibration and light). In other words, app's functionality is provided to users via SPS.

Next, let us see *how privacy disclosures may provide functionality to apps*. For a privacy disclosure through network requests, after received by the remote server, network response(s) may be returned to the app to fulfill app's functionality in a certain way. For example, as shown in Listing 1, after disclosing private data `DeviceID` and `phone number`, the app receives a response (line 15-21) and displays it to user through UI (line 22-25). However, if there is no response at all or only invariant responses are received by the app even if the disclosed private data has changed, then this privacy disclosure will not affect any functionality of the app. In other words, the app may put arbitrary data instead of user's private data in the requests without affecting the functionality of the app, and users will not experience any difference. Hence, if changing certain private data in a privacy disclosure

does not cause the change of network responses (no response is a special case of no change of response), we consider it a privacy leak (i.e., unjustifiable).

The next question is: *if the network responses change with privacy disclosures, will this prove the privacy disclosures are necessary and thus justifiable*? The answer is: not necessarily. If the response of a privacy disclosure, even though changed, cannot lead to any human sensible phone states (SPS) directly or indirectly, users will not be able to experience the difference. Hence, we still consider this privacy disclosure unjustifiable. For example, as shown in Listing 2, a snippet code from the GoldDream malware sample discloses several types of private data (including DeviceId, SubscribeId, and SimSerialNum, in lines 1-5,), then gets a `uid` from a remote server (lines 6-9), and finally stores it locally (lines 10-12). This uid will be used to uniquely identify the user on the server side and will change as these private data change. However, since the response of the privacy disclosure cannot lead to any SPS, it cannot be experienced by users and hence the privacy disclosure corresponding to the response is unjustifiable. Indeed, in this example, disclosing these private data is not necessary - the app can instead derive a uid locally based on these private data and transmit the uid to the remote server.

Figure 1 shows the overall workflow of our approach, which mainly consists of three steps. **Step 1.** We borrow the request differential analysis technique from AGRIGENTO [12] to detect privacy disclosures. **Step 2.** We propose the technique of *response differential analysis* to test whether the responses change with disclosed private data. If changes are not observed, we consider the detected privacy disclosure as an unjustifiable privacy leak. Consider the privacy disclosure containing `DeviceID` and `phone number` (Listing 1), when we change these private data one by one, we found that only changes of `DeviceID` caused responses to change whereas changes of `phone number` did not. Thus, while this privacy disclosure may partially affect app's function, as a whole it is a privacy leak. **Step 3.** If response changes are observed, we further perform *response taint analysis* to determine whether such responses serve app's functions. More specifically, we will check whether a response handling API will be linked to any SPS-related API through static taint flow analysis. If true, it is an justifiable privacy disclosure; otherwise, it is an unjustifiable privacy leak. Consider the example in Listing 2, through response differential analysis we found the received `UID` changed with disclosed private data. However, no data flow was found from the response handling API to any SPS-related API. Thus, this privacy disclosure is still considered as a privacy leak.

As mentioned above, to detect privacy disclosures, we perform request differential analysis as in AGRIGENTO [12]. It works in two steps: first, it establishes a baseline of network request behavior of an app through multiple dynamic executions (also called *first-phase* executions in Section 4.3.1); second, by modifying certain private data and running this app again (also called *second-phase* executions in Section 4.3.1), it identifies the requests with changes as privacy disclosures.

Challenges in Diagnosing Privacy Leaks. As shown in Figure 1, although it may appear at first glance to be straightforward to perform diagnosis on detected privacy disclosures, we need to solve two important challenges during the diagnosis: ***1)*** For each privacy disclosure detected from *second-phase* execution, to perform *response differential analysis*, we need to compare its response with the corresponding responses generated during *first-phase* executions and determine whether the response has changed or not. Since multiple *first-phase* executions have generated many requests and responses, here the challenge is to accurately identify the responses corresponding to the request causing the privacy disclosure. To this end, we propose to first identify the program statement generating this privacy disclosure (i.e., transmitting the request). Then, from all requests captured during *first-phase* executions, we identify the requests which are generated by the same program statement and consider their responses as the ones corresponding to this privacy disclosure. ***2)*** For each privacy disclosure, to perform *response taint analysis*, we will use each program statement that receives network responses as a source and the SPS related APIs as sinks. The challenge here is to locate the response-receiving program statement corresponding to each privacy disclosure. To handle the above two problems, we develop a new technique,

named *network pair identification*, and integrate it with *bytecode instrumentation* to locate program statements of privacy disclosure and its response. More details can be found in Section 4.1 and 4.2.

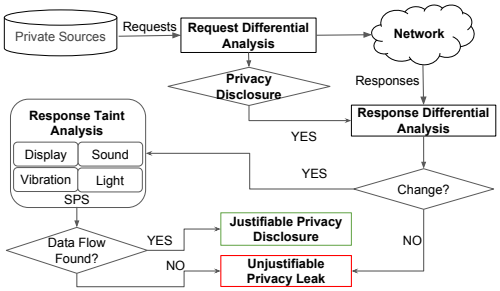


Fig. 1. High-level workflow. Request differential analysis is used to detect privacy disclosure through network; response differential analysis and taint analysis are combined to perform diagnosis by judging whether the detected privacy disclosure serves app's functions.

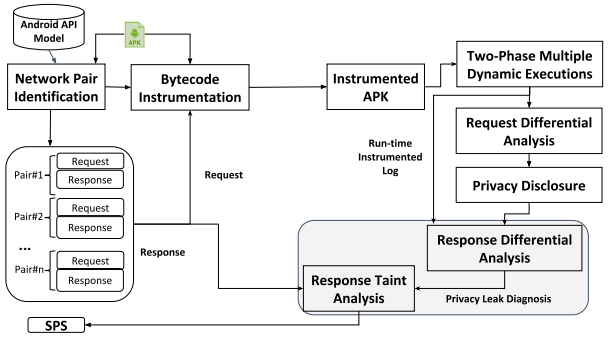


Fig. 2. Overall Architecture of LeakDoctor. First, network pair identification (section 4.1) and bytecode instrumentation (section 4.2) are used to support the following response differential analysis (section 4.3.3) and taint analysis (section 4.4) by generating run-time log; second, prior to request and response differential analysis, two phase multiple dynamic executions (section 4.3.1) are needed.

4 SYSTEM DESIGN DETAILS

Figure 2 shows the overall architecture of our system design. Next, we describe the details of each component.

4.1 Network Pair Identification

To assist the following privacy leak diagnosis, we propose to locate the underlying program statements of privacy disclosures. First, we identify all the possible network pairs through static analysis. A network pair **NP** refers to a pair of program statements involved in a network transaction, one sending out a network request and the other handling its response. The intuition here is that privacy disclosures must be generated by some program statements, and these statements are usually based on well-known Android and Java APIs. Through manually analyzing and collecting known network-related APIs, we come up with three general types of network pairs, as shown in Table 1 (due to space limit, we only show well-known cases):

1). The first case is synchronous network request and response implemented through `HttpURLConnection`. In this case, we pair network request and response through static taint flow analysis. For instance, as shown in Listing 1, we use `URL`(line 9) as source, response statement (line 15) as sink and perform taint propagation. In the end, we collect a taint path: 9-11-15 and identify line 11 (request) and line 15(response) as a network pair.

2). The second case is synchronous network request and response implemented through various `execute()` methods, such as `HttpClient.execute()`. In these cases, aided by the known Android API model, we directly scan the program code to identify the related statements as network pairs.

3). The third case is asynchronous network request and response, where requests are implemented by handler methods (e.g. `enqueue()`), while responses are processed by callback methods (e.g. `OnResponse()`). In these cases, we first discover the request handler methods by scanning program code, and then adopt points-to analysis to get their response callback methods and identify them as pairs.

Table 1. Known Network Pair Cases

Pair Type	Sample Cases	Inserted Statements
Synchronous	Request: HttpURLConnection.OutputStream(); Response: HttpURLConnection.InputStream()	HttpURLConnection. getURL()
	Request and Response: HttpClient.execute()	HttpClient.getURI().toURL()
Asynchronous	Request: OkHttp.Call.enqueue(); Response: OnResponse()	OkHttp.Request.url()

For each identified network pair, we will assign an identifier `ID` to represent this pair and record the related statements. This information is needed to perform response taint analysis in a later time.

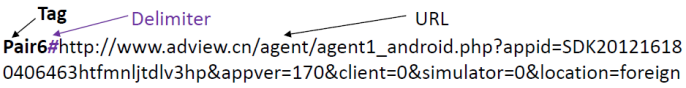


Fig. 3. Run-Time Log Example with URL and Pair ID

4.2 Bytecode Instrumentation

Given the network request and response pairs identified from the last step, we then insert several statements to dump the run-time `URL` information of each request. Consider the example shown in Listing 1 again, the detail steps are as follows. First, a network request (line 11) and response (line 15) pair is identified from the previous step and its pair `ID` is allocated as 'Pair1'. Second, a new statement is inserted to capture the `URL` field of this network request, as listed in the third column of Table 1. Since the request in this example is generated through building a `HttpURLConnection`, we capture its `URL` information by inserting a statement invoking the `getURL()` method of `HttpURLConnection`, which is shown in line 11*. Third, to dump the `URL` information locally, another `Log` statement is inserted by calling a system API, whose 'TAG' is the pair `ID` and 'Content' is the captured `URL`, as shown in line 12*. Finally, the instrumented APK will be executed during dynamic analysis and a run-time log will be generated. Figure 3 shows an example, where a request's `URL` and its pair `ID` are recorded. Aided by the run-time log, for each detected privacy disclosure (i.e., request), we can first get its network pair `ID` by extracting its `URL` information, and locate the related program statements based on this `ID` in a later time.

4.3 Request and Response Differential Analysis

Figure 4 shows the overview of differential analysis, including both request and response differential analysis. Prior to differential analysis, an app needs to go through two-phase dynamic executions.

4.3.1 Two-Phase Dynamic Executions. **In the first phase**, an app is executed multiple times to collect raw network traces and construct a network request behavior summary. As stated in [12], the key prerequisite for performing request differential analysis is to eliminate any sources of non-determinism between different executions. Only so can we reliably attribute any changes in the network request (as a result of changing private input data) to privacy disclosures. Here, as in [12], we run the app in an instrumented environment, which can record various contextual information of the app during each run, including random values, timing values, encrypted values, etc. Then the collected contextual information is used to remove the non-determinism in the network request traces of all first-phase runs, and help construct the overall network request behavior summary for this app, shown as solid lines in Figure 4. The final network request behavior summary is modeled using a four-layer tree-based structure [12], which contains the network request flows from all the collected network traces. The first layer of the tree contains all the `domain` names of captured request flows, and the second layer contains the paths recorded for each `domain`. The third and fourth layers contain key-value pairs from the header

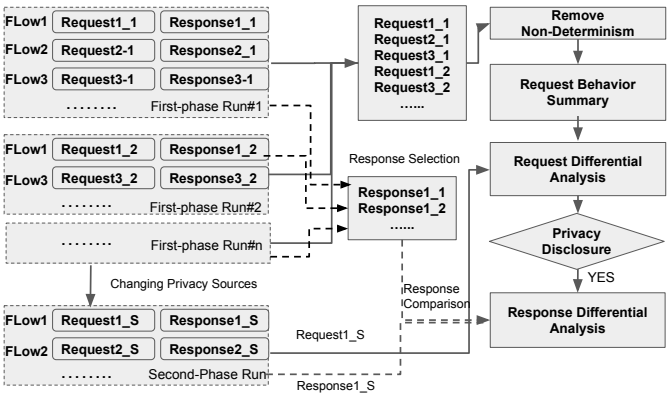


Fig. 4. Overview of Differential Analysis. Each captured request or response is annotated with suffix *i_j*, where *i* indicates the corresponding flow number. *j* indicates the run number in the first phase when *j*=1,2,..., while *j*=*S* indicates specifically a request or response captured in the second phase.

and payload of http requests. This structure is useful to group the fields of http requests according to their 'types' and positions in the tree. It is also convenient to compare the fields of different network requests.

A distinctive aspect of the first phase is how it determines the number of times each app should be executed. As in work [12], after each run, LeakDoctor performs request differential analysis using the collected traffic and contextual information. By analyzing the discrepancies in the network request behavior without changing any source of private information, we can know when LeakDoctor has sufficiently explored the app's network request behavior, i.e., when the network request behavior summary has reached convergence. In practice, we say an app has reached convergence when we do not see any discrepancies in the request behavior summary for K consecutive runs. It has been shown in [12] that $K = 3$ provides the best trade-off. Also, since some apps might take very long time to reach convergence, we set a maximum number of runs (15).

In the second phase, we run the app in the same instrumented environment, but we selectively modify the value of private data (e.g., IMEI, location) we want to track. Specifically, we will change all *at once* the values of all private data of our interest to quickly detect if an app has any privacy disclosure. We will also change them one-by-one – each time only changing one privacy source – to precisely identify the type of private information the app is disclosing and perform privacy leak diagnosis for each type.

4.3.2 Request Differential Analysis. For each request flow captured in the second-phase run, we adopt the differential analysis algorithm [12] to compare it against the network request behavior summary tree constructed in the first phase. We consider it as a privacy disclosure if it changes after private data modification. In practice, we first determine whether an app has any privacy disclosure through the **change-all-at-once** execution. If the app has no privacy disclosure, the analysis will terminate. Otherwise, **change-one-by-one** executions are performed for fine-grained privacy disclosure detection and privacy leak diagnosis.

4.3.3 Response Differential Analysis. Here, we assume *Request1_S* shown in Figure 4 is a privacy disclosure and use it as an example to illustrate the process of response differential analysis on its response, following Algorithm 1. To determine whether the response of *Request1_S* has changed or not after we modify certain private data, we first perform *response selection* to collect the responses of the same requests from the first-phase runs as

well as its response from a second-phase run, and then compare these collected responses through a *response comparison* process.

Responses Selection. At the first step, the response of *Request1_S*, namely *Response1_S*, in the second-phase run is extracted using URL as the identifier (lines 6-11 in Algorithm 1). Next, the responses of the same requests as *Request1_S* from first-phase runs need to be extracted. However, since there are also many responses from other different requests, it is impractical to do this by simply using URL as identifier. For example, being a privacy disclosure case, private data modification always causes the URL of same requests to change between first-phase runs and second-phase run. To accurately select the responses of same requests instead of other requests from first-phase runs, we rely on the run-time log that was generated earlier. According to the *URL* of *Request1_S*, we get its network pair info **np** from the run-time instrumented log (line 14). Based on **np**, we collect the responses of same requests which belong to the same network pair from the first-phase runs (lines 15-20), namely *Response1_1*, *Response1_2*, etc. We do this because same requests must be sent from the same program statement.

ALGORITHM 1: Response Differential Analysis.

```
1  Flows_b =All network flows collected from first-phase runs;
2  Flows_a =All network flows collected from second-phase run;
3  Function Response Differential Analysis(PrivacyDisclosure)
4      before_res ← {};                                   // Responses selected from Flows_b
5      after_res ← {};                                    // Responses selected from Flows_a
6      url ← ExtractURL(PrivacyDisclosure);
7      foreach flow ∈ Flows_a do                          // Extract Response of PrivacyDisclosure
8          url2 ← ExtractURL(flow);
9          if url == url2 then
10             response ← ExtractResponse(flow);
11             after_res = after_res ∪ response;
12         end
13     end
14     network_pair1 ← getPairInfo(url);                  // //from the run-time log
15     foreach flow ∈ Flows_b do                          // Response selection from first-phase runs
16         url1 ← ExtractURL(flow);
17         network_pair2 ← getPairInfo(url1);             // //from the run-time log
18         if network_pair1 == network_pair2 then
19             response ← ExtractResponse(flow);
20             before_res = before_res ∪ response;
21         end
22     end
23     return Compare(before_res, after_res)
24 Function Compare(before_res, after_res)
25     Map1 ← ExtractField_Value(before_res);
26     ChangeFields ← ExtractChangeField(Map1);
27     Map2 ← ExtractField_Value(after_res);
28     foreach field, value ∈ Map2 do
29         if field ∉ Map1 then
30             return True
31         end
32         else
33             if field ∉ ChangeFields and Map1(field) ≠ Map2(field) then
34                 return True
35             end
36         end
37     end
38     return False
```

Response Comparison. After the responses are selected, response comparison is implemented following the procedure Compare (lines 24-38) defined in Algorithm 1. Similar to request differential analysis, the key for

Table 2. Android framework APIs that can change SPS

Type	Method Name	Quantity
display	setText,setTitle,setIcon, etc	237
sound	setDataSource, setSound, etc	12
vibration	setVibrate, vibrate	4
light	setLights	2

accurately performing responses comparison is to eliminate the noise in responses, mainly including fields with frequently changing values. Since we cannot acquire and analyze the server-side logic of an app, it is not easy to detect and remove all noise in responses. To address this issue, we adopt a heuristic approach. First, we define the noise in responses as the fields which always change values. For example, 'Date' field is widely found in network responses, but its value always changes. Second, we try to extract the noise from the responses, which is collected from multiple first-phase runs. The intuition here is that if the value of a same field of responses collected from the first-phase runs always changes even without private data modification, we can ascertain that it will also change after private data modification. Consequently, we consider this field as a non-determinism. Due to such non-determinism, responses will always show changes, which prevent us from accurately attributing response changes in the second-phase run to the changes of private data. Therefore, we need to extract noise and filter them away before response comparison.

We use the example mentioned above to illustrate the process. To extract the noise, we parse the headers and payloads of each http response selected from first-phase runs, namely *Response1_1*, *Response1_2*, etc and store them into a field-value map **M** (line 25). During the parsing phase, the known data structures (e.g., `JSON`) are also recognized, parsed and stored into the same field-value map. In addition, response decryption is also implemented when needed, by hooking Android Crypto APIs. A detailed description can be found in Section 5. If the value of a field in the extracted field-value map **M** always changes, we consider this field as a noise (line 26) and filter it out during the following comparison.

After extracting the noise from the selected responses, the response (i.e.,*Response1_S*) of *Request1_S* is also parsed into a field-value map **m** (line 27) in the same way. Then, the comparison between **M** and **m** is performed on both the field and value levels (line 28-37). If the return value of this comparison is true, it means that the response of *Request1_S* has changed after private data modification.

4.4 Response Taint Analysis

After dynamic response differential analysis, we can collect privacy disclosures whose disclosed private data' changes will cause corresponding responses to change. To determine further whether such a privacy disclosure will serve certain functionality of the app, LeakDoctor adopts static taint analysis to link its response with potential SPS-related APIs. For this purpose, aided by the instrumented run-time log, we can get the network pair `ID` for this privacy disclosure and then locate the corresponding response program statements. It is worth mentioning that dynamic response differential analysis help us a lot and reduce the processing overhead of static taint flow analysis by starting from a few deterministic response points (i.e., collected responses after differential analysis) instead of all potential response points. Second, LeakDoctor uses the located program statements as sources and 255 SPS-relevant framework APIs as sinks. Most of these sinks were previously collected by [10], as shown in Table 2. For example, `TextView:void setText`(CharSequence) is a framework API to change the display of a text editor widget. Finally, an inter-procedural data-flow analysis is implemented. If a data flow is found, the corresponding response is considered to change the SPS and this privacy disclosure is considered necessary to serve certain functions of the app.

Since network responses usually affect SPS in both direct and indirect manner, two common cases need to be solved. First, to handle indirect data flows from response to SPS through data medium (e.g., SharedPreference),

LeakDoctor performs an additional two-stage indirect data flow analysis: from response sources to data medium sinks, and from correlated data medium sources to SPS sinks. Inspired by the fact that data mediums are always accessed by using a string as identifier, we adopt constant propagation and points-to analysis technique to correlate all potential data medium sinks and data medium sources. If a data medium sink and a data medium source share same identifier, they are correlated and their corresponding sub-flows will join together to form a complete data flow. Second, to handle data flows through Inter-Component Communication (ICC), LeakDoctor reuses techniques proposed in [30] and [39] to track the data flows from responses to SPS through known ICC methods.

Note that compared to dynamic taint flow analysis, such as TaintDroid [14] and TaintART [53], static taint flow analysis may be less precise when linking response with SPS. However, currently both TaintDroid and TaintART use a list of APIs as sources and do not differentiate the context of API invocations. This may lead to some errors for our analysis as we need to differentiate specific program statements (those directly handling responses) as sources for taint tracking. In this sense, FlowDroid fits our need better. Of course, static analysis and dynamic analysis each has its well-known strengths and limitations, so combining them could give us better results, which we leave as a part of future work.

5 IMPLEMENTATION

In this section, we provide some implementation details of LeakDoctor, including the tools it is built upon and some key components. The whole system is written in Java and Python, consisting of 10,378 LOC.

LeakDoctor extends Flowdroid [4] to perform static taint analysis in both network pair identification and response taint analysis. As in [3], we adopt Soot to implement bytecode instrumentation on the Jimple level. We also modify Flowdroid to find the known network pairs by building known network API models. More specifically, inspired by [27], we build these models by manually selecting high-level Java and Android APIs or callbacks that are commonly used for network request and response processing from related classes and http libraries, including *org.apache.http*, *java.net*, *okhttp*, *Volley*, *retrofit*, etc.

To capture network traffic, we reuse the techniques proposed in [12]. Specifically, LeakDoctor captures the `HTTP` traffic and inspects `GET` and `POST` requests using a proxy based on the mitmproxy library [2]. In order to intercept `HTTPS` traffic through man-in-the-middle, we install a CA certificate on the instrumented device. Furthermore, to be able to capture traffic when apps use certificate pinning, we install JustTrustMe [1] on the device, which is an Xposed module that disables certificate checking by hooking the related Android APIs that perform this check (e.g., `getTrustManagers()`). Finally, to filter out only the network traffic generated by the analyzed app, `iptables` is used to mark packets generated by the `UID` of the app, and route only those packets to our proxy.

In order to decrypt encrypted responses, we hook the Android Crypto APIs (i.e., `Cipher`,etc) and store the arguments and return values of each method. Based on the fact that if a network response is encrypted by a remote server, it must be decrypted by client-side code before further processing. Thus, we parse the API traces to build a decryption map that allows it to map a ciphertext to the corresponding original plaintext.

To generate the run-time log, we hook the `Log` system API through an Xposed module. As did in [12], we currently only track and actively modify eight kinds of privacy sources: `AndroidID`, `contacts`, `ICCID`, `IMEI`, `location`, `MAC address`, `IMSI`, and `phone number`.

6 EVALUATION

In this section, we evaluate the effectiveness and efficiency of LeakDoctor, and focus on the following questions:

- **Question 1**: How accurate and effective is LeakDoctor to assist analysts diagnosing privacy leaks?
- **Question 2**: How does LeakDoctor perform compared with the closely related work DroidJust?

Table 3. Privacy Disclosure Detection and Privacy Leak Diagnosis Results of LeakDoctor

Initial Privacy Disclosure Detection		Our Automated Privacy Leak Diagnosis		Manual Verification (**TPs**)	
#Apps with Privacy Disclosures	377	#Apps with Privacy Leaks	263	#Apps with Privacy Leaks	255
#Total Privacy Disclosures	2095	#Total Privacy Leaks	588	#Total Privacy Leaks	561

- **Question 3**: What new applications can LeakDoctor be used for?
- **Question 4**: What is the performance overhead incurred by LeakDoctor?

6.1 Experiment Setup

We perform our experiments on two Nexus 5 phones running Android 4.4.4, while LeakDoctor is deployed on a 16GB RAM, 4-core machine running Ubuntu 16.04. The devices and the machine running LeakDoctor were connected to the same subnet, allowing LeakDoctor to capture the generated network traffic. By using real devices instead of emulators, our evaluation is more realistic. For each execution, we run an app by using Monkey for UI stimulation. We provide Monkey with a fixed seed so that its interactions are the same across multiple runs. Although the fixed seed is not enough to remove all randomness from the UI interactions, it helps eliminate most of randomness. At the end of each run, we uninstall the app and delete all of its data.

6.2 Datasets

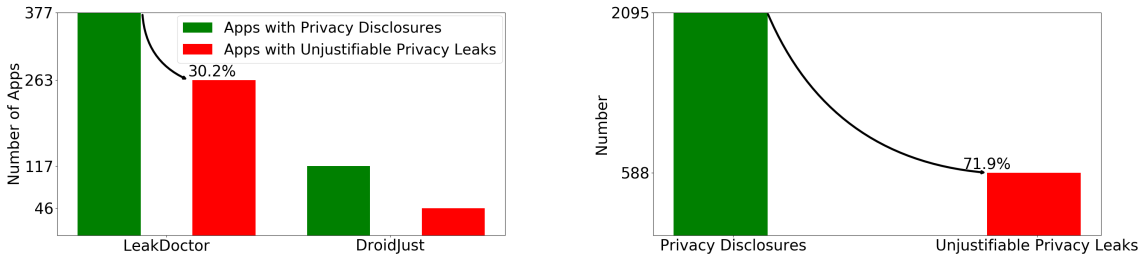
We conduct our experiments on 1060 apps, 850 of which were used in the evaluation of AGRIGENTO [12]. They include 750 apps from the Recon [44] dataset and top 100 apps from Google Play. Another 210 apps were collected from Anzhi (a leading third-party app market in China).

As shown in Table 3, in the first step, 2095 privacy disclosures were detected from 377 apps through request differential analysis. Note that our privacy disclosure detection is based on AGRIGENTO [12] and it has been well evaluated. Since our goal is to diagnose privacy leaks, the following evaluations will focus on privacy leaks. Generally, performing evaluation on privacy leak diganosis is far from trivial, the main problem being the absence of ground truth. Therefore, we manually reverse engineered the apps to our best ability to confirm the results. Here, we call it a true positive(TP) when LeakDoctor correctly reports a privacy leak (or correctly reports a privacy-leaking app in the app level measurement). On the contrary, we call it a false positive (FP) when LeakDoctor incorrectly reports a justifiable privacy disclosure as a privacy leak (or incorrectly reporting such an app in the app level measurement). The concepts of false negative (FN) and true negative (TN) are also obvious. To measure the effectiveness of LeakDoctor, we adopt the metric **Accuracy** =(TP+TN)/(TP+FN+TN+FP).

6.3 Evaluation on Privacy Leak Diagnosis

6.3.1 Results Analysis. As shown in Table 3, among the 2095 privacy disclosures detected from the 377 apps, LeakDoctor reports 588 privacy leaks from 263 apps through privacy leak diagnosis. To distinguish true positives from false positives, we manually verify the detected privacy leaks.

Manual Verification. For each detected privacy leak by LeakDoctor, we consider it as a *false positive* when it does not leak any private data or its response changes during response differential analysis and also has data flow to certain SPS. Accordingly, the manual verification consists of three steps. 1) Check whether it indeed leaks at least one kind of private source; 2) Inspect its response traffic to confirm that the responses have changed in response differential analysis; 3) Inspect the related program code paths to confirm no data flow from its response to a SPS. Compared to the first two steps, the validation in the third step is more difficult. As a best effort, we assume that Flowdroid, which we use, can discover all direct data flows with no false negatives, and we mainly focus on two kinds of indirect data flows solved by ourselves. First, aided by the taint flow analysis



(a) App-Level Privacy Disclosure Detection and Diagnosis.

(b) Privacy Disclosure Detection and Diagnosis of LeakDoctor.

Fig. 5. Privacy Disclosure Detection and Pirvacy Leak Diagnosis of LeakDoctor and DroidJust.

results of Flowdroid, we collect all the medium sinks (e.g., sendBroadcast(),insert()), which can be reached by the response of a reported privacy leak, and all the medium sources (e.g., getIntent(),query()), which can change the SPS. Second, we manually check among the collected medium sinks and medium sources, any two of them can be correlated when they represent the same components or storage locations. If yes, we consider there is an undiscovered data flow from the response of this privacy leak to SPS, that is, a false positive case.

Finally, we are able to manually verify that 561 privacy leaks from 255 apps are true positives, and the remaining 27 privacy leaks and 8 apps detected by LeakDoctor are false positives. We manually analyze these false positive cases in detail and summarize two root causes below. 1) LeakDoctor uses AGRIGENTO for privacy disclosure detection, while AGRIGENTO fails to handle some sources of non-determinism and incorrectly report some requests as privacy disclosures. 2) Our current static response taint analysis cannot track all the response data flows as we expected. The remaining 14 false positives are due to this reason. For example, *com.gtp.nextlauncher.theme.gs4* processes network response through ICC with complicated string operations. LeakDoctor shares the same limitation with IccTA [30] and fails to solve this kind of cases. When app *com.dianxin os.dxbs-4.1.9* stores and reads responses through SharedPreference with non-constant string identifiers, LeakDoctor also fails to handle.

To assess the false negatives of LeakDoctor, we manually inspect the remaining 1507 privacy disclosures, which are considered justifiable by LeakDoctor. Here, we consider a privacy disclosure as a *false negative* when its response does not change during response differential analysis or has no data flow to any SPS. For each privacy disclosure, the manual checking process was straightforward to perform since LeakDoctor has output its results of response differential analysis (i.e., response changes) and response taint flow analysis (i.e., the taint flow paths). Finally, We did not find any false negatives during manual checking. We acknowledge that it is not possible to exclude the possibility of false negatives through manual verification. However, we believe that our results are encouraging towards automated privacy leak Diagnosis.

6.3.2 Accuracy and Effectiveness. As shown above, LeakDoctor reports 588 privacy leaks from 263 apps, among which 255 apps and 561 privacy leaks (true positives) from the initial detection results (377 apps and 2095 privacy disclosures) reported by AGRIGENTO. That is, it can identify apps containing privacy leaks with an accuracy 97.9%, and detect privacy leaks with an accuracy 98.7% and with false positive rate 1.8% and no false negative. If we consider our manually verified privacy leaks and apps as the ground truth, AGRIGENTO generated 1534 more false positives in terms of number of leaks and 122 more false positives in the app level.

Based on the above comparison, we can conclude that LeakDoctor can effectively diagnose privacy leaks and related apps. As shown in Figure 5, it can exclude 114 apps (30.2%) and 1507 privacy disclosures (71.9%) because

they are justifiable by app's functionality. This can help privacy analysts avoid firstly analyzing these justifiable cases and mainly focus on the true privacy leaks. Note that 'justifiable' is not equal to legitimate. Of course, privacy analysts can also perform further checks on these justifiable privacy disclosures based on the diagnosis results provided by LeakDoctor.

6.4 Comparison with DroidJust

To further evaluate our approach, we perform a comparison with DroidJust [10], a very relevant approach. Since the definition of privacy disclosure in DroidJust is different from ours, we consider leaks only at the app level. We run DroidJust on our entire dataset (1060 apps). As shown in Figure 5(a), it identifies 117 apps with privacy disclosures and through its justification, it reports that 46 of them contain privacy leaks. However, LeakDoctor has correctly detected 255 apps with privacy leaks previously.

By comparing the detection results of these two tools, we find that 24 apps are detected by both LeakDoctor and DroidJust as containing privacy leaks. In addition, LeakDoctor detects 231 apps and DroidJust detects 22 apps exclusively. We then manually check the 22 apps exclusively reported by DroidJust and find 14 of them have not generated any network traffic during dynamic analysis. Because DroidJust detects privacy disclosures through static taint analysis, as long as the code is not overly obfuscated, it can report most (if not all) cases. On the other hand, since LeakDoctor detects privacy disclosures through request differential analysis with AGRIGENTO [12], whose performance is also affected by the effectiveness of UI exploration, these 14 cases are potential false negatives in LeakDoctor. For the remaining 8 apps, by manually checking their source code and results of DroidJust, we find that they contain privacy disclosures, whose related responses lead to SPS through ICC. Since DroidJust is unable to track data flow through ICC, it incorrectly labels these privacy disclosures as unjustifiable privacy leaks and generates these false positives.

Moreover, we manually verify the 231 privacy-leaking apps detected exclusively by LeakDoctor. Among them, 53 cannot be analyzed by DroidJust due to insufficient memory or failure on type resolving when detecting privacy disclosures. Basically, this is a common problem in many FlowDroid-dependent tools, which conduct static taint flow analysis. LeakDoctor avoids this problem well by relying on dynamic differential analysis to detect privacy disclosures. At the meantime, dynamic differential analysis help narrow the scope of 'sources' used in the static response taint analysis (i.e., response points with changes) and reduce the processing overhead a lot. Another 121 apps contain privacy leaks using various obfuscation and encryption techniques, and hence cannot be detected and justified by DroidJust through static taint flow analysis. The remaining 57 apps also contain privacy leaks, by sending out network requests containing *both* necessary and unnecessary private data for app's functionality, as in case study 1 (Section 6.7). By static taint analysis only, DroidJust often cannot accurately match network responses with network requests, especially when there are multiple network flows. Moreover, when it justifies a network request with a response, it simply justifies all the data fields contained in the request and hence incurs false negatives. Lack of fine-grained analysis is a fundamental limitation in DroidJust. Aided by the dynamic response differential analysis, LeakDoctor can correctly discover that these privacy disclosures partially contain private data unnecessary for app's functionality and hence, they are still unjustifiable.

The above evaluation shows that compared to DroidJust, through integrating dynamic differential analysis and static taint analysis, LeakDoctor is more effective and accurate to identify apps with privacy leaks.

6.5 Application 1: Data Access Control on Individual Privacy Data Source

To protect user's privacy, researchers have proposed to generate mock data to be sent to the network, such as in AppFence [23], MockDroid [6], SmartPer [40], etc. To this end, they are required to first accurately determine which kind of privacy data source is not necessary for app's functionality and hence should be mocked. However,

this requirement was not addressed by these approaches. In this work, by changing private data one by one, LeakDoctor can not only detect privacy disclosure from a certain privacy source, but also help determine which kind of private data is necessary to disclose. To demonstrate this, we study 30 samples, which contain privacy leaks according to LeakDoctor. As shown in Table 4, LeakDoctor performs privacy leak diagnosis on individual privacy source with 40 true positives and 8 false positives in total and it shows a high true positive rate for all individual privacy sources. [2] In this way, by combining with LeakDoctor, previous works can leverage the privacy leak diagnosis results on each kind of privacy source and protect user's privacy by mocking these unnecessarily leaked private data.

Table 4. Privacy Leak Diagnosis on Each Privacy Source

Package Name	P1	P2	P3	P4	P5	P6	P7	P8
com.easy.battery.saver	①	①	1	0	0	0	0	0
me.lyft.android-3.50.1.63651	①	1	0	0	0	0	0	0
com.jb.emoji.gokeyboard-2.75	1	①	0	①	0	0	0	0
com.recycle.bin	①	1	①	①	0	0	0	0
com.mob4.virtualcompass	1	1	①	0	0	0	0	0
jp.co.translimit.braindots	1	①	0	1	①	0	1	0
mobi.yellow.booster1.1.24.apk	1	1	①	①	0	1	0	0
com.nexon.hit.global-1.1.77191	1	1	①	①	1	1	0	0
com.devuni.flashlight	①	1	0	0	0	0	0	0
com.ss.android.article.master-2.1.2	1	1	①	0	0	0	0	0
com.outfit7.mytalkingangelafree	①	1	0	0	①	0	0	0
com.jb.zcamera-2.30	0	1	1	①	0	0	0	0
com.hulu.plus-2.25.1.202755	0	1	0	①	①	0	0	0
com.donopo.mary10	1	1	①	0	0	①	0	0
com.xime.latin.lite-1.0.11	1	1	0	0	①	0	0	0
air.com.sgn.juicejam.gp	①	0	1	1	0	1	0	1
com.webascender.callerid	0	①	0	0	0	0	1	0
com.staircase3.opensignal	0	1	0	0	0	①	0	0
com.life360.android.safetymapd	①	1	0	0	1	0	0	1
com.namcobandaigames	1	1	0	①	①	①	0	①
com.nordcurrent.canteenhd-2.0.1	1	①	0	1	0	0	0	0
com.zynga.words	1	①	0	0	0	0	0	0
com.smule.singandroid-3.8.5	①	1	0	0	0	0	0	0
com.enflick.android.TextNow	1	0	0	0	①	①	0	0
com.fgol.HungrySharkEvolution	1	1	0	①	0	0	0	0
com.ijinshan.kbatterydoctor-en	1	①	①	1	0	0	0	0
com.jb.gosms-7.09	1	①	0	①	①	0	0	0
app.tools.media.photoeditor	1	①	0	0	0	0	0	0
mobi.infolife.cache	①	0	0	1	0	0	0	0
com.christmasgame.appleshoot	1	①	0	0	0	0	0	0
TPs	8	10	4	7	6	4	0	1
FPs	1	0	3	2	1	0	0	1

P1: Android ID, P2: IMEI, P3: Location, P4:Mac Address, P5: IMSI, P6: ICCID, P7: Contacts, P8: Phone Number. '0' indicates it is not a privacy disclosure, '1' indicates it is a true privacy disclosure, and the '①' indicates it is a privacy leak, as diagnosed by LeakDoctor.

Table 5. Privacy Leak Diagnosis Results on Known Libraries Used in Dataset

Type	Name	#apps	#J_PD	#U_PL	P_S
Analytics	Flurry	28	4	59	P1, P2, P3, P8
	Umeng	16	10	18	P2
	Appbrain	7	3	18	P1, P2, P3, P4
	Unity 3D	5	1	31	P2, P3, P4
	Adjust	4	2	8	P1, P2,P7
Advertising	AppsFlyer	23	23	47	P1, P2,P3
	ChartBoost	15	66	0	
	AppLovin	10	15	13	P3, P5, P7, P8
	Tapjoy	9	58	18	P1, P2, P4, P7
	InMobi	7	10	10	P1
Total		124	192	222	

#J_PD indicates justifiable privacy disclosures from third party library; #U_PL indicates unjustifiable privacy leaks from third party library; P_S indicates unjustifiably leaked privacy sources; P1,P2,etc, are defined as same in Table 4.

6.6 Application 2: Privacy Leak Diagnosis on Known Third-Party Libraries

Although many previous works [22, 55] have revealed various privacy leaks caused by third-party libraries, they cannot differentiate the necessary privacy disclosures from unnecessary ones. In this work, we leverage LeakDoctor's capability of privacy leak diagnosis to study privacy leak issues in third-party libraries.

To evaluate this, we perform an in-depth evaluation on the unjustifiable privacy leaks detected by LeakDoctor on the entire dataset. By extracting the network `domain` (e.g., *.com) in the network request for each detected

[2]We notice that LeakDoctor produces relatively high false positives for some specific private sources (e.g., location and phone number), because our request differential analysis, currently based on AGRIGENTO [12], fails to properly handle some sources of non-determinism.

privacy leak, we can determine where it's generated, either from app program logic itself or from known third-party libraries. In this way, we make statistics over all unjustifiable privacy leaks detected from our dataset and collect the top-10 third-party libraries that contain privacy leaks. Here, it is worth mentioning that AGRIGENTO [12] which LeakDoctor uses was set to filter out (i.e., whitelist) Google Ads. On one hand, because they contain non-determinism which AGRIGENTO cannot efficiently eliminate at the current time and thus cause false positives. On the other hand, the traffic to Google was considered less privacy risky. Therefore, we did not perform privacy leak diagnosis on Google Ads in our context.

As shown in Table 5, totally, 124 apps are found using the collected top-10 third-party libraries. Of the 414 privacy disclosures generated by these third libraries, 222 are diagnosed as privacy leaks by LeakDoctor. Based on these results, several findings can be drawn.

Finding1. From Table 3 and 5, among the total 561 privacy leaks identified by LeakDoctor, 39.6% (222/561) originate from third-party libraries and 60.4% from app's main logic. As such, distinguishing between privacy disclosure and privacy leaks only based on whether they are from third libraries or not is inaccurate. Especially, some applications choose to leak privacy quietly and maliciously through their own program logic instead of using some known third libraries. LeakDoctor is helpful to detect such privacy leaks.

Finding2. As shown in Table 5, 46.4% (192/414) privacy disclosures generated from third-party libraries are justifiable, indicating that not every privacy disclosure caused by third libraries is a privacy leak. However, we also notice that they tend to disclose more than necessary. For example, the *Tapjoy* library contains a privacy disclosure, which leaks `android-id`, `device-id` and `mac address` in a single request to update advertisements. This privacy disclosure is justifiable in DroidJust [10] as it does not differentiate multiple data fields in one request. However, through response differential analysis, we find that only `device-id` can cause the related response to change, so only `device-id` is necessary for app's functionality; in other words, neither of `android-id` and `mac address` should be disclosed for app's functionality. For this reason, this privacy disclosure as a whole is unjustifiable according to LeakDoctor. Note that for privacy advocates who consider targeted advertising in an app as undesirable or as functionality irrelevant, LeakDoctor can be easily configured to directly exclude ads libraries for diagnosis (e.g., output any privacy disclosure through ad libraries as privacy leaks without further analysis).

Finding3. As shown in Table 5, compared to advertising libraries, the ratio of privacy leaks caused by analytics libraries is higher. This is due to their different purposes. Analytics libraries are mainly used to collect and analyze the interactions between users and apps for helping developers to gain insights about users' behaviors. However, oftentimes these privacy disclosures do not directly serve app's functionality. Therefore, according to our definition, most of them are diagnosed as privacy leaks. On the other hand, given the specific purpose of analytics libraries, one may choose to white-list some known benign ones during privacy leak diagnosis.

6.7 Case Studies

We manually reverse engineer some apps that are automatically diagnosed as containing privacy leaks by LeakDoctor. Here, we present two case studies.

Case study 1. As shown in Figure 6, the *com.b2creativedesigns.eyetest* app leaks several privacy sources using obfuscation. It first puts the `IMEI`, `IMSI`, and `Phone Number` in a `JSON` object. The `JSON` object is then encrypted using `AES`, encoded using `Base64` and sent to a remote server. The response is a confirmation message which updates a text component on the screen. Since it updates the UI and serves some functionality, according to DrodiJust [10] and [45], this privacy disclosure will be considered as justifiable. However, through response differential analysis, we find that only changes in `IMEI` will cause the response to change. Thus, the `IMSI` and `phone number` are unnecessary to disclose for app's functionality. This indicates that response differential analysis in LeakDoctor helps identify privacy leaks that previous work [10, 45] fails to identify.

Fig. 6. Network Requests from Two Apps

Case study 2. Although most privacy leaks can be identified by response differential analysis, some of them need to be examined further through response taint analysis. For example, the *com.dianxinos.optimizer.duplayvb* app leaks `IMEI` and `android ID` through a request, as shown in Figure 6. Our response differential analysis shows that changes in either `IMEI` or `android ID` cause the received response to change ('request-id' field value changes from '1089749834' to '2022684510' and '2094450351', respectively). However, through the follow-up response taint analysis, we observe that the response of this privacy disclosure does not update any SPS. Therefore, this privacy disclosure does not serve app's functionality and hence is unjustifiable. This indicates that response differential analysis alone is not sufficient to identify all privacy leaks, and response taint analysis is also indispensable.

6.8 Performance Overhead Evaluation

To evaluate the performance overhead of LeakDoctor, we make the evaluation on the entire dataset (1060 samples) and record the analysis time of apps with privacy disclosures. LeakDoctor mainly includes four components: namely network pair identification, bytecode instrumentation, multiple dynamic executions and privacy leak diagnosis. Recall that the process of multiple dynamic executions in LeakDoctor is the same as in [12], which has been well evaluated. On average, an app is executed 13 times in about 98.8 minutes, network pair identification takes 24.2 seconds, bytecode instrumentation 31.2 seconds and privacy leak diagnosis 155.9 seconds. In total, each app can be analyzed in 102.3 minutes on average, among which the multiple dynamic execution process takes 96.6% of the overall time. As stated in [12], this process can easily scale up using multiple devices running the same app in parallel. The above results show that our method is capable of large-scale offline analysis, especially when parallel processing is available with multiple devices.

7 DISCUSSION

7.1 Design Considerations

As stated in [17, 25], most users cannot be relied on to effectively protect their privacy since they may have varying opinions and understandings of privacy and even inconsistencies between their attitudes on privacy and their actual behavior in response to privacy issues. To help protect user's privacy, we choose to place our focus on diagnosing privacy leaks in mobile apps in advance before they were distributed to users. Thus, we proposed LeakDoctor to help developers, app store owners and security analysts understand whether a privacy disclosure does serve some app's functionality or not. Based on the diagnosis results of LeakDoctor, there may be several application scenarios through which user's privacy could be protected better. **First**, the app store owners may ask developers to explain the existence of unjustifiable privacy leaks in their newly submitted apps and protect

users from installing the apps with unexplained and unjustifiable privacy leaks. **Second**, app developers may avoid using certain third-party libraries with more unjustifiable privacy leaks and further protect user's privacy in the application development phase. **Third**, if necessary, appropriate privacy protection recommendations could also be provided to users and assist them use other tools to mock or block certain private data [6, 11].

As mentioned before, some apps are designed for special functions, including location-tracking, back up, etc. For back up apps, they may send out private information quitely to the remote cloud disk through network (e.g., uploading photos to iCloud). In these cases, privacy disclosures are often designed to be one-way (i.e., no responses with rich returned information and only with simple acknowledged information '200 OK') and will be labelled as privacy leaks by LeakDoctor. While this sounds like false positive cases, technically speaking LeakDoctor works correctly as intended. It is only a different interpretation of diagnosis results in specific contexts, not a fundamental flaw of LeakDoctor. Previous works WHYPER [41] and AutoCog [42] were proposed to automatically infer an app's functions and necessary permissions from its description by using natural language processing. These approaches may extract additional useful information from the apps' descriptions and assist security analysts easily distinguish such cases.

7.2 Limitations

As we explore a new approach for diagnosing privacy leaks, our work inevitably has a number of limitations.

Some complex privacy disclosure cases through network are not covered: As described above, LeakDoctor borrows the techniques in [12] to perform multiple dynamic executions and hence it inherits some limitations from those techniques, which may cause some potential false negatives. First, the limited code coverage of Monkey may result in some privacy disclosures not being detected (i.e., false negatives). To solve this, we are planing to improve the code coverage through more intelligent and directed test generators [7, 36]. Second, LeakDoctor still suffers from some covert channels [9] by which an attacker could leak information without being detected. Actually, the differential analysis based method severely limits the bandwidth of the channel an attacker can use to stealthily leak private data [12]. Third, LeakDoctor is limited by the number of protocols it currently tracks: it only examines the `GET` and `POST` requests of `HTTP` and `HTTPS` (intercepted through man-in-the-middle proxy even with certificate pinning in most cases). We leave the extension of LeakDoctor to handle other protocols as our future work.

Static network pair identification may fail: In some cases, our current prototype may fail to locate the program statements of captured network traffic. First, the current implementation mainly handles several known `HTTP` cases during network pair extraction and bytecode instrumentation. The network traffic generated by other less-known network libraries currently has not been correlated with the related program code. Second, network traffic can also be generated by `Javascript` code executed in a WebView, which sometimes happens in case of ad libraries. Since our bytecode instrumentation works at bytecode level and does not instrument `Javascript` code, currently we cannot locate the program statements of network traffic generated by `Javascript` code.

Static response taint analysis may be bypassed: Although we have tried to perform precise response taint analysis, it still has some limitations. First, like most static analysis methods, static response taint analysis may fail to track some complex data flows, which may lead to false positives. Second, theoretically it is possible for a malicious app to circumvent this by introducing noisy inbound network flows which only cause minimal changes of SPS, thus causing false negatives for LeakDoctor. We consider such an attack is possible for determined attackers although we have not observed any app in our experiment with such behavior. However, there are practical challenges for this attack. First, the forged response by this attack would most likely be considered as noise and hence ignored during our analysis. Recall that LeakDoctor performs response differential analysis based on two-phase dynamic execution, as stated in Section 4.3.3. In the first phase, LeakDoctor keeps the private data unchanged and tries to identify the non-determinism field (i.e., noise) in the responses through multiple

runs. The intuition behind this is that, if the value of the same field of responses collected from the first-phase runs always changes even without private data modification, we can ascertain that it will also change after private data modification. Consequently, we consider this field as a non-determinism noise and filter it away during the response comparison after changing private data in the second phase. For a malicious app, it is hard to detect that it is being analyzed by LeakDoctor and manifests differently. Thus, if it tries to generate dummy changes to SPSs every time it tries to gather sensitive data, the dummy changes will be considered as noise and be filtered without having any affect on LeakDoctor. Note that this attack is effective against DroidJust, because, based on static analysis, it cannot tell the changes of message contents.

Finally, we believe that the design philosophy of LeakDoctor has made such attacks more difficult to succeed: the attacker not only needs to redesign server logic to add response changes in response to our active changes in private data, but also needs to ensure the response changes lead to certain SPS (e.g., make a tiny change in the interface that is not noticeable). In this way, LeakDoctor has raised the bar to the evasion attempts. In the meantime, further research is certainly needed to examine the practicality of this attack. As did in [45], a potential solution is that we can capture the screenshots and analyze UI changes in a dynamic manner. By ignoring certain insignificant differences, this method can be more resilient than static taint analysis.

Code obfuscation and native code may bypass static analysis: The static modules of LeakDoctor did not consider various code obfuscation techniques (e.g., reflection, packing, native code, etc), which aim to make certain code unavailable to static analysis. With code obfuscation, network related identifiers and control flow may be modified and even hidden from the source code, making LeakDoctor impossible to perform static network pair identification and bytecode instrumentation, let alone the following response differential analysis and response taint analysis. Thus, although LeakDoctor could adopt Agrigento[12] to detect privacy disclosures originating from obfuscated code or native code, it still will fail to perform justification on each detected privacy disclosures. To address this issue, one solution is to deobfuscate the app based on the code structure that cannot be obfuscated. Fortunately, recently much progress has been made to solve the code obfuscation issue [13, 31, 51, 54], which is orthogonal and complementary to our method.

False positives caused by dynamic response differential analysis: As stated before, after actively and randomly changing private data, LeakDoctor relies on dynamic response differential analysis to determine whether the response of a privacy disclosure changes or not. However, if a remote server returns a different response only when the request data falls into a specific category, false positives may happen. Because, in this case, randomly modifying private data is hard to trigger this different response returned by the server. Theoretically speaking, this problem may be mitigated by sending a large number of requests with different private data. There is a tradeoff since sending more requests could result in more responses, which may introduce noise affecting the response comparison. Further investigation is needed to examine this kind of false positives.

7.3 Future Work

As stated before, it is an implementation choice for LeakDoctor to adopt AGRIGENTO [12] to detect privacy disclosures and perform further privacy leak diagnosis. Actually, LeakDoctor's diagnosis method is generic. Recently, another work MUTAFLOW [38] was proposed to detect privacy disclosures through mutating private data. MUTAFLOW systematically mutates dynamic values returned by privacy sources to assess whether the mutation changes the values passed to sensitive sinks. If so, a flow between source and sink (i.e., privacy disclosure) is found. Similar to AGRIGENTO, this mutation-based flow analysis of MUTAFLOW does not attempt to identify the specific path of the information flow and is thus also resilient to obfuscation. As a part of future work we would explore MUTAFLOW as front-end of LeakDoctor to detect privacy disclosures.

Additionally, We have adapted AGRIGENTO [12] on Android 7.1 by updating some related APIs. As part of our future work, we intend to perform some evaluation on the new version of Android.

Currently, LeakDoctor determines the origin of a privacy leak (app or third-party library code) by using the network domain name. This method will not work with obscure/less-known/obfuscated domains. Recently, many works, such as LibD [32], LibPecker [58], have been proposed to identify the third-party libraries within apps. Therefore, they can be adopted to solve the domain name problem. Aided by bytecode instrumentation, we may insert extra monitoring code and locate the specific class and package where privacy leak occurs. Based on the third-party library detection results from LibD [32] or LibPecker [58], LeakDoctor can further determine whether its identified class with privacy leak is from a third-party library. For example, LibPecker adopts signature matching to perform an obfuscation-resilient, highly precise and reliable library detection. By fully utilizing the internal class dependencies inside a library, LibPecker generates a strict signature for each class. Thus, based on the detection results from LibPecker, LeakDoctor can accurately determine the privacy leak originated from a third-party library. We also leave this enhancement as our future work.

As far as we know, the current methods can only determine that certain private information was leaked, but cannot automatically reveal how it was leaked. In this work, we have proposed several techniques to locate program statements of privacy disclosures. Based on such knowledge, we will characterize how a privacy disclosure is implemented (e.g., by obfuscation) by integrating static program slicing and semantic analysis [27].

8 RELATED WORK

In the literature, many approaches have been proposed to detecting privacy leaks and enhancing privacy protection for users. They roughly fall into four categories.

Static Analysis. AndroidLeaks [20] was one of the first static taint analysis approaches, but it lacks precision as it tracks data flow at the object-level. MorphDroid [16] tracks atomic units of private information to account for partial leaks. FlowDroid [4] is a precise context, flow, field, object-sensitive and lifecycle-aware static taint analysis tool to detect privacy leak. Additional approaches include EdgeMiner [8], which addresses the issue of reconstructing implicit control flow transitions, Amandroid [52], and IccTA [30], which deal with inter-component data leaks. AppAudit [55] addresses the false positives of static taint analysis by verifying leaks through approximated dynamic execution. All these prior works have primarily considered privacy leaks originating at the client directed at the server, while leakage in the reverse direction – from the server to the client – is comparatively under-studied. To answer this question, SIFON [28] is proposed to analyze web APIs to determine the extent of oversharing of user information where the server sends information to the client app that is never used. Unlike these static analysis based methods aiming to detect privacy disclosure, PrivacyStreams [33] includes a library and a static analyzer to ease app developers' programming efforts on dealing with personal data in their apps. Furthermore, using PrivacyStreams to access and process personal information allows data auditors and end-users to better understand the privacy implications of mobile apps by exposing the detailed processing steps and data granularity of personal data accesses.

Dynamic Analysis. Dynamic taint information flow analysis tracks data as it is being processed during runtime. TaintDroid [14] is a dynamic analysis tool and uses a modified Dalvik virtual machine for tracking information flow between private sources and sinks. BayesDroid [49] is similar to TaintDroid, but it uses probabilistic reasoning to classify a leak based on the similarity between the data at both points. TaintART [48] extends TaintDroid to native code. A most recent work MUTAFLOW [38] proposes to detect information flow through mutating input data. Unlike many other techniques, mutation-based flow analysis by MUTAFLOW does not attempt to identify the specific path of the flow and thus is resilient to obfuscation. Similar to TaintDroid, MockDroid [6] proposes to modify Android and provide users with the ability to "mock" sensitive data accessed by applications at runtime. AppFence [23] extends TaintDroid to mitigate the privacy leak problem by dynamically shadowing user specified sensitive data and blocking exfiltration of private data over the network. PmP [11] follows a similar runtime enforcement technique, but focuses on informing users about access requests from

third-party libraries and supports a novel feature of both app- and library-based privacy control for accesses to sensitive data. As stated by our evaluation, some popular third-party libraries indeed need to disclose some private data to provide rich functionality for users but they tend to disclose more than necessary. However, PmP [11] cannot solve this case. By leveraging the privacy leak diagnosis results of LeakDoctor, the previous work can mock or block these unnecessarily leaked private data.

Network Traffic Interception. Several recent works have been proposed to detect privacy leak at the network level, usually by capturing network traffic from device through a virtual private network (VPN) tunnel and detecting privacy leaks on the fly. Tools such as AntMonitor [29] and PrivacyGuard [47], perform their analysis on-device using Android's build-in VPNService. ReCon [44] is another VPN-based approach, which uses a machine learning classifier to identify leaks, but it cannot handle many obfuscation-based cases. Agrigento [12] aims to detect privacy leaks by performing black-box differential analysis and is more resilient to obfuscation.

Permission and Privacy Disclosure Purposes. Most of the aforementioned approaches only focus on privacy disclosure detection, regardless of the legitimacy of disclosure. However, security analysts may want to understand the purpose of privacy disclosures and related permission requests. CHABADA [21] is proposed to automatically infer an app's necessary permissions from its description. [50] also concentrates on inferring the possible purpose behind a permission request via static analysis to educate users and enable them to make better decisions. Similarly, PERUIM [34] proposes to relate user interface with permission requests through program analysis and adopt permission-UI mapping as an easy-to-understand representation to illustrate how permissions are used by different UI components within a given application. However, all of these three works cannot be directly used to justify privacy leaks because their focus is at permission level instead of information flow level. AppIntent [57] first stresses the necessity to justify the legitimacy of privacy leak, but needs too much human effort. MudFlow [5] tries to detect abnormal privacy leak by learning "normal" application behavior patterns of used sources and sinks, and identifying outliers using machine learning. COSMOS [19] is a context-aware mediation system that learns the expected behavior of an app in each context from a large training dataset. It bridges the semantic gap between foreground interaction and background access, in order to protect system integrity and user privacy. The effectiveness of MudFlow and COSMOS highly depends on the training data sets.

AsDroid [24] is proposed to identify contradictions between a user interaction and the behavior that it performs and it assumes that all operations are triggered by the UI, which is not applicable for all privacy disclosure cases. AAPL [37] proposes to identify legitimate privacy disclosures through a peer-voting mechanism and assumes that apps in the same category do imply they have the same functionality. This assumption leads to relatively high false positive rate and false negative rate. [45] proposes to detect covert communication by determining whether an external communication has effect on the user-observable application functionality. Its main limitation is that it only justifies communication that affects application UI in a direct manner rather than indirect case, such as through data medium, while LeakDoctor can solve both cases. DroidJust [10] tries to justify the legitimacy of privacy disclosures by correlating privacy disclosures with SPS. Based on static taint analysis only, it has difficulty to correlate requests and responses accurately when multiple network flows exist and cannot differentiate multiple data fields in requests and responses. As a result, it incurs high false negatives, as shown in the comparative study in Section 6.4. LeakDoctor overcome its limitations by using dynamic response differential analysis. In [56], based on the analysis result of DroidJust, the authors designed SweetDroid, a dynamic access control system to enforce context-sensitive privacy policies for Android apps. Compared to DroidJust, LeakDoctor can generate more accurate privacy policies for SweetDroid to enforce.

9 CONCLUSION

In this paper, we proposed a fine-grained approach named *LeakDoctor* to automatically justify whether each privacy disclosure in an app serves app's functionality. Unlike to most existing works, LeakDoctor works on

information flow level, and can differentiate the necessary privacy disclosures from unnecessary ones, i.e., unjustifiable privacy leaks. To this end, LeakDoctor integrates request differential analysis and response differential analysis, proposes new techniques for network pair identification, and handles indirect information flow and ICC problems for response taint analysis. Our evaluation on 1060 apps with 2095 privacy disclosures shows that LeakDoctor can automatically diagnose privacy leaks with a high accuracy (98.7%). It identified 71.9% of 2,095 privacy disclosures as justifiable. Hence, LeakDoctor can help analysts quickly focus on the true privacy leak threats. Additionally, our evaluation results show that it can be applied to analyze third-party libraries and assist mocking-based data privacy protection mechanisms.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. Zhu's work was supported in part by National Science Foundation (NSF) under grants CNS-1618684. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencie.

REFERENCES

[1] 2016. JustTrustMe. *Online: https://github.com/Fuzion24/JustTrustMe*. (2016).
[2] 2016. mitmproxy. *Online: https://mitmproxy.org*. (2016).
[3] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Instrumenting Android and Java Applications as Easy as abc. In *International Conference on Runtime Verification*. 364–381.
[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick Mcdaniel. 2014. FlowDroid:precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
[5] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 426–436.
[6] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. 2011. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*. ACM, 49–54.
[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*.
[8] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Network and Distributed System Security Symposium*.
[9] Swarup Chandra, Zhiqiang Lin, Ashish Kundu, and Latifur Khan. 2014. Towards a systematic study of the covert channel attacks in smartphones. In *International Conference on Security and Privacy in Communication Systems*. Springer, 427–435.
[10] Xin Chen and Sencun Zhu. 2015. DroidJust: automated functionality aware privacy leakage analysis for Android applications. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*.
[11] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I Hong, and Yuvraj Agarwal. 2017. Does this App Really Need My Location?: Context-Aware Privacy Management for Smartphones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* UbiComp'17, 3 (2017), 42–63.
[12] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Network and Distributed System Security Symposium*.
[13] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and X Wang. 2018. Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In *25th Annual Network and Distributed System Security Symposium, NDSS*. 18–21.
[14] William Enck, Peter Gilbert, Byung Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick Mcdaniel, and Anmol N. Sheth. 2010. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Acm Transactions on Computer Systems* 32, 2 (2010), 1–29.
[15] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 627–638.

[16] Pietro Ferrara, Omer Tripp, and Marco Pistoia. 2015. Morphdroid: fine-grained privacy verification. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 371–380.
[17] Denzil Ferreira, Vassilis Kostakos, Alastair R Beresford, Janne Lindqvist, and Anind K Dey. 2015. Securacy: an empirical investigation of Android applications' network usage, privacy and security. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 11.
[18] Hao Fu, Zizhan Zheng, Somdutta Bose, Matt Bishop, and Prasant Mohapatra. 2017. Leaksemantic: Identifying abnormal sensitive network transmissions in mobile applications. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*. IEEE, 1–9.
[19] H. Fu, Z. Zheng, S. Zhu, and P. Mohapatra. 2019. Keeping Context In Mind: Automating Mobile App Access Control with User Interface Inspection. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*.
[20] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. *Trust* 12 (2012), 291–307.
[21] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1025–1035.
[22] Jiang X Grace M, Zhou W. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. 101–112.
[23] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 2011. 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*. ACM, 639–652.
[24] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1036–1046.
[25] Corey Brian Jackson and Yang Wang. 2018. Addressing The Privacy Paradox through Personalized Privacy Notifications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 2 (2018), 68.
[26] Thivya Kandappu, Archan Misra, Shih-Fen Cheng, Randy Tandriansyah, and Hoong Chuin Lau. 2018. Obfuscation at-source: Privacy in context-aware mobile crowd-sourcing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 1 (2018), 16.
[27] Jeongmin Kim, Hyunwoo Choi, Hun Namkung, Woohyun Choi, Byungkwon Choi, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han. 2016. Enabling Automatic Protocol Behavior Analysis for Android Applications. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 281–295.
[28] William Koch, Abdelberi Chaabane, Manuel Egele, William Robertson, and Engin Kirda. 2017. Semi-automated discovery of server-based information oversharing vulnerabilities in Android applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 147–157.
[29] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. 2015. AntMonitor: A System for Monitoring from Mobile Devices. In *ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big*. 15–20.
[30] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *IEEE International Conference on Software Engineering*. 280–291.
[31] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 318–329.
[32] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: scalable and precise third-party library detection in android markets. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 335–346.
[33] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I Hong. 2017. PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* UbiComp'17, 3 (2017), 76.
[34] Yuanchun Li, Yao Guo, and Xiangqun Chen. 2016. Peruim: Understanding mobile application privacy with permission-ui mapping. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 682–693.
[35] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, S Aerin Zhang, Norman Sadeh, Y Agarwal, and A Acquisti. 2016. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Symposium on Usable Privacy and Security*.
[36] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 643–653.
[37] Kangjie Lu, Zhichun Li, Vasileios P. Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. 2015. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Dataflow Analysis and Peer Voting. In *2015 Network and Distributed System Security Symposium*.

[38] Björn Mathis, Vitalii Avdiienko, Ezekiel O Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting information flow by mutating input data. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 263–273.

[39] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 77–88.

[40] Katarzyna Olejnik, Italo Ivan Dacosta Petrocelli, Joana Catarina Soares Machado, Kévin Huguenin, Mohammad Emtiyaz Khan, and Jean-Pierre Hubaux. 2017. SmarPer: Context-Aware and Automatic Runtime-Permissions for Mobile Devices. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*. IEEE.

[41] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications.. In *USENIX Security Symposium*. 527–542.

[42] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1354–1365.

[43] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: automatic security analysis of smartphone applications. In *ACM Conference on Data and Application Security and Privacy*. 209–220.

[44] Jingjing Ren, Martina Lindorfer, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *International Conference on Mobile Systems, Applications, and Services*. 361–374.

[45] Julia Rubin, Michael I Gordon, Nguyen Nguyen, and Martin Rinard. 2015. Covert communication in mobile applications (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 647–657.

[46] Nazir Saleheen, Supriyo Chakraborty, Nasir Ali, Md Mahbubur Rahman, Syed Monowar Hossain, Rummana Bari, Eugene Buder, Mani Srivastava, and Santosh Kumar. 2016. mSieve: differential behavioral privacy in time series of mobile sensor data. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 706–717.

[47] Yihang Song and Urs Hengartner. 2015. Privacyguard: A vpn-based platform to detect information leakage on android devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 15–26.

[48] Mingshen Sun, Tao Wei, and John Lui. 2016. 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*. ACM, 331–342.

[49] Omer Tripp and Julia Rubin. 2014. A Bayesian approach to privacy enforcement in smartphones. In *Usenix Conference on Security Symposium*. 175–190.

[50] Haoyu Wang, Jason Hong, and Yao Guo. 2015. Using text mining to infer the purpose of permission use in mobile apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 1107–1118.

[51] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1137–1150.

[52] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. 311 (2014), 1329–1341.

[53] Tao Wei, Tao Wei, and John C. S. Lui. 2016. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *ACM Sigsac Conference on Computer and Communications Security*. 331–342.

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

[55] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. 2015. Effective Real-Time Android Application Auditing. In *Security and Privacy*. 899–914.

[56] Chen Xin, Huang Heqing, Zhu Sencun, Li Qing, and Guan Quanlong. 2017. SweetDroid: Toward a Context-Sensitive Privacy Policy Enforcement Framework for Android OS. In *Proceedings of ACM Workshop on Privacy in the Electronic Society (WPES), in conjunction with the ACM CCS conference*. ACM.

[57] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *ACM Sigsac Conference on Computer and Communications Security*. 1043–1054.

[58] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 141–152.

Received August 2018, revised November 2018, accepted January 2019