Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting

Kangjie Lu[1], Zhichun Li[2], Vasileios P. Kemerlis[3], Zhenyu Wu[2], Long Lu[4],
Cong Zheng[1], Zhiyun Qian[5], Wenke Lee[1], Guofei Jiang[2]
[1]Georgia Institute of Technology, [2]NEC Labs America, Inc., [3]Columbia University,
[4]Stony Brook University, [5]University of California Riverside
[1]{kjlu, cong}@gatech.edu, wenke@cc.gatech.edu, [2]{zhichun, adamwu, gfj}@nec-labs.com,
[3]vpk@cs.columbia.edu, [4]long@cs.stonybrook.edu, [5]zhiyunq@cs.ucr.edu

***Abstract*— Serious concerns have been raised about stealthy disclosures of private user data in smartphone apps, and recent research efforts in mobile security have studied various mechanisms to detect privacy disclosures. However, existing approaches are not effective in informing users and security analysts about potential privacy leakage threats. This is because these methods largely fail to 1) provide highly accurate and inclusive detection of privacy disclosures, and 2) filter out legitimate privacy disclosures that usually dominate detection results and in turn obscure true threats.**

In this paper, we propose AAPL, an automated system that detects privacy leaks (*i.e.,* truly suspicious privacy disclosures) in Android apps. AAPL is based on multiple special static analysis techniques that we've developed for Android apps, including conditional flow identification and joint flow tracking. Furthermore, AAPL employs a new approach called peer voting to filter out most of the legitimate privacy disclosures from the results, purifying the detection results for automatic and easy interpretation.

We implemented AAPL and evaluated it over 40,456 apps. The results indicate that, on average, AAPL achieves an accuracy of 88.7%. For particular disclosures (e.g., contacts), the accuracy is up to 94.6%. Using AAPL, we successfully revealed a collection of unknown privacy leaks. The throughput of our privacy disclosure analysis module is 4.5 apps per minute on a three-machine cluster.

I. Introduction

The growth of smartphone application (*i.e.,* app) markets have been truly astonishing, as reflected in the ever increasing user population and the constantly enriching offering of apps that virtually satisfy all forms of digital needs of the users. As apps are used for more and more private and privileged tasks by the users, concerns are also rising about the consequences of failure to protect or respect user's privacy (*i.e.,* transferring it to remote entities or publicizing it). As a result, many approaches have been proposed to automatically uncover privacy disclosures in Android Apps, falling into two major categories: static control and data flow analysis [1], [14], [18], [20], [26], [40], and dynamic data flow tracking [13], [31].

NDSS '15, 8-11 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2015.23287

Although previous works successfully revealed the pervasiveness of privacy disclosures in apps and made significant progress towards the automatic detection of privacy disclosures, two major shortcomings remain to be addressed: (1) relatively low coverage of data-flows; (2) incapability of judging the legitimacy of detected flows. The first shortcoming prevents current data-flow analysis from identifying complex data-flows, such as conditional or joint flows, which are frequently seen in Android apps. Conditional flows are unique to Android apps and caused by generic system APIs that can access a variety of data sources and sinks, including sensitive ones (*e.g.,* contacts and sensitive content providers), which are determined solely by the runtime parameters (*e.g.,* URIs and flags). Joint flows consist of two or more sub-flows, implicitly connected outside of app code (*e.g.,* inside database, file system, or OS), which may form a channel at runtime and disclose private data. The second shortcoming often results in inflated detection results containing too many false alerts (*e.g.,* benign or functional privacy disclosures). These alerts usually do not represent violations of user privacy, and therefore, distract or even overwhelm human users and analysts when interpreting detection results. Our study shows that more than 67% app privacy disclosures found using conventional methods are in fact legitimate (*i.e.,* necessary to apps' core functionalities). For example, navigation apps need to report user's current location to remote servers for up-to-date maps and real-time road conditions. From now on, we use the term *privacy disclosure* to generally describe apps' actions that propagate private data to external entities. We reserve the term *privacy leak* only for describing a privacy disclosure that cannot be intuitively justified by the app's intended functionaries.

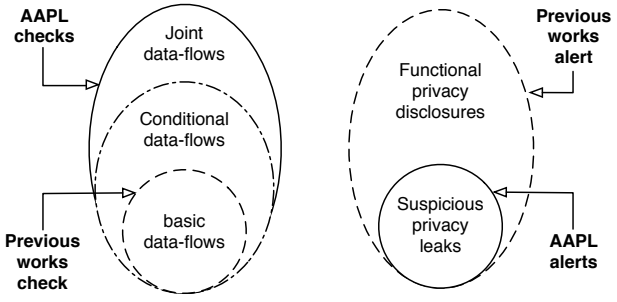

Fig. 1. Improvements of AAPL over previous works

In this paper, we propose AAPL (Analysis of App Privacy Leak, pronounced as "Apple") to address both of the short-

comings discussed above. As the left part of Figure 1 shows, AAPL improves the state-of-art data-flow analysis by adding the capability of identifying and checking the complex flows. These complex flows constitute several types of common privacy disclosures in Android apps. Including these flows significantly increases the detection rate of privacy disclosures. Furthermore, unlike previous works, which report all detectable privacy disclosures, AAPL is capable of judging the legitimacy of privacy disclosures and only alerting privacy leaks, which are the true concerns of app users and security analysts.

AAPL achieves highly accurate detection by introducing three new techniques to existing static data-flow analysis of Android apps: opportunistic constant evaluation, object origin inference, and joint flow tracking. The first two techniques overcome the open research challenge of correctly recognizing conditional sources and sinks of sensitive data and identifying the related data-flows. The third technique allows AAPL to form joint data-flows from seemingly disconnected data-flows that are connected by external code during runtime. These improvements better adapt existing static analysis techniques to the unique programming paradigms of Android apps and therefore allow AAPL to detect the data-flows that are invisible to existing methods.

Once the improved data-flows analysis has detected privacy disclosures, AAPL continues to discover privacy leaks using a novel *peer voting mechanism*. This mechanism is inspired by the fact that applications with the same or similar functionality (*i.e.,* peer apps) should exhibit similar privacy consumption behaviors, and therefore incur a similar set of privacy disclosures. The peer voting mechanism determines the legitimacy of a particular privacy disclosure detected in an app (*i.e.,* primary app) by consulting the privacy disclosure profiles of its peer apps. If the privacy disclosure is common among the peer apps (*i.e.,* majority of the peers "vote" for it), it is considered to be necessary to the primary app's main functionalities. Otherwise, the privacy disclosure is likely to be caused by dispensable or unexpected features of the primary app, and therefore represents a high risk of privacy leak. The peer app collection process is fairly straightforward. For a primary app, AAPL queries the existing app recommendation systems. Such systems, either an integral feature of app stores or a standalone service by 3rd parties, returns a list of apps that are functionally similar or related to the primary app. However, it is not uncommon to see apps of different functionalities returned on the same list by these systems. This is because apps are considered as related not only when they're functionally similar but also when they are frequently installed or used together (*e.g.,* a restaurant review app and a movie review app). AAPL filters out these minor noises by applying basic semantic similarity analysis on app descriptions and removing apps whose descriptions have a long semantical distance from the descriptions of other peers.

We evaluated AAPL using $2,777$ popular apps as primary apps, which led to $37,679$ peer apps in total. We randomly chose 417 primary apps for in-depth evaluations, which involve 532 unique privacy disclosures. AAPL can achieve a 88.7% accuracy with a false positive rate of 10.7% and a false negative rate of 12.5%. For some specific leaks (*e.g.,* contacts), AAPL can achieve an even better accuracy of 94.6% with a false positive rate of 4.4% and a false negative rate of 9.9%.

This paper makes the following main contributions:

- We improve the accuracy and coverage of privacy disclosure analysis using multiple techniques, including conditional flow identification through opportunistic constant evaluation and object origin inference, and joint flow tracking. Results show that the enhancements increase detection rate by 31% while reducing the false positive rate by a factor of 5.
- We propose using a novel peer voting mechanism to automatically differentiate the security-critical privacy leaks from the more general legitimate privacy disclosures.
- We evaluate AAPL on $40,456$ apps. The results show AAPL has a high accuracy and high performance.

The remainder of this paper is organized as follows. Section II overviews of the privacy disclosure problem and AAPL system. Section III and Section IV present the design and implementation of AAPL. Section V reports the evaluation results. We discuss the limitations in Section VI and related work in Section VII. Finally, we conclude the paper in Section VIII.

II. Overview

As mobile users continue to rely on apps for personalized services or business mobility, apps are increasingly entrusted with more and more private and sensitive information. Meanwhile, a large number of apps without functional dependency on user data also use (monetize on) user privacy to varying degrees, ranging from typically benign cases, like targeted advertising, to ill-intended ones, like identity theft. As a result, mobile users on one hand are largely in favor of personalized services, but on the other hand become more and more concerned about apps abusing their data. This issue is worsened by the current lack of tools or methods to inform users of potentially harmful privacy leaks in their apps without distracting or confusing users with apps' legitimate privacy disclosures.

Mainstream smartphone OSs, such as Android and iOS, provide basic protection on user sensitive information, such as the permission system, which enforces coarse-grained and static access control on sensitive resources as per users' explicit consent. However, success of such systems largely assumes a users' common awareness and sometimes deep understanding on the privacy impacts of an app's advertised features. In practice, this assumption is not true. Moreover, these systems offer little clues and help to users when soliciting their consent.

A. Privacy Disclosure Analysis

To help users make more informed decisions, many existing privacy disclosure detection systems were built to reveal apps' behaviors that propagate sensitive data (*i.e.,* a source) to a sensitive channel (*i.e.,* a sink) [13], [26]. These systems perform either static or dynamic data-flow analysis on apps. Dynamic approaches, such as TaintDroid [13], feature low false positives, while the static detection approaches, such as CHEX [26], achieve high coverage and scalability. Works

following either approaches have demonstrated effective detection of privacy disclosures, and their results [12], [26], [31], [39] show that about 20%-40% of all apps in Google Play market disclose user privacy for different purposes and through different channels.

However, simply reporting the existence of privacy disclosures may not be meaningful enough to app users or security analysts. Even if the privacy disclosure of an app is detected and presented users, the burden of comprehending each privacy disclosure of an app and deciding its negative privacy impact can be untenable for the average users. Let us consider a common privacy disclosure through which a user's location is sent to the Internet. Current detection systems alert users about such privacy disclosures whenever they are detected in an app, regardless of if the app's core functionalities depend on them. Therefore, whenever analyzing a legitimate navigation app or a calculator app with a 3rd-party library aggressively tracking users, current detection systems alert users with the same type of reports that indicate potential privacy violations.

B. Determining Disclosure Legitimacy via Peer Voting

In our work, we aim to provide a solution that can automatically infer if a given privacy disclosure is legitimate or not (*i.e.,* whether it is likely required by an app's core functionalities). Considering the large amount of competing apps with similar functionalities, we form the hypothesis that one can derive the legitimate privacy disclosures required by the core functionalities across a collection of similar app. On the other hand, privacy disclosures that are uncommonly used by a collection of similar apps would naturally stand out, and are likely unrelated to the advertised core functionalities of the app. Based on this hypothesis, it is possible to build a system which can automatically screens all privacy disclosures, filters out those which are likely related to app's core functionalities, and only flags the highly suspicious privacy disclosures.

To properly implement and verify our hypothesis, we face three challenges: 1) We need some way to find functionally similar apps. As will be described later in Section III, we mostly leverage existing app recommendation systems such as the one provided by Google Play, and employ a natural language processing (NLP) technique called semantic similarity [32] to further purify similar apps. 2) We need a highly accurate analysis system that can capture privacy disclosure flows accommodating with the special Android programming paradigm. As will be described later in Section III-A, we choose static analysis since it allows us to discover more complete privacy disclosure flows with high performance. 3) We need an automated approach to differentiate the highly suspicious privacy disclosures from legitimate ones. Our approach is shown in Section III-B.

C. AAPL Usage Scenarios

AAPL is designed to be practical and efficient so that it can be operated by 1) large-scale app market providers, to screen their hosted apps; 2) users, to avoid apps that do not respect their privacy; 3) developers, to understand the privacy issues of the third-party libraries they use. AAPL is a general privacy leak flow detection system that can be taken advantage of by multiple principles. First, AAPL could be taken as an efficient detection system for market providers to detect the apps with potential suspicious privacy disclosures. To further determine whether the detected suspicious privacy disclosures are real privacy leaks, market providers can challenge developers to justify why and how privacy disclosures are used in the app's which functionality. Failure to justify the privacy disclosures might result in rejection of the app from the markets. Without our system, simply challenging every privacy disclosure is unacceptable, as 1) most privacy disclosures are actually legitimate; 2) verifying the justifications of large-scale apps is resource-consuming for market providers. AAPL aims to detect highly suspicious privacy disclosures, and only a small portion of AAPL detection results are false positives that mainly caused by special functionalities of an app (*e.g.,* messaging in the photo editor app). Challenging the developers of such apps is acceptable. Second, AAPL can help privacy-concerned users identify the apps with suspicious privacy disclosures. Users usually do not know why and how the privacy disclosures are used in the app. Simply reporting the privacy disclosures to users provides limited help to understanding how well the app respects users' privacy. AAPL provides an important metric for users to understand how likely the privacy disclosure is a privacy leak, and the suspicious privacy leaks when comparing with similar apps. Third, developers can make use of AAPL to check whether their apps have suspicious privacy disclosures. If the suspicious privacy disclosure is caused by third-party libraries, the developer can choose an alternative library to avoid suspicious privacy disclosures. On the other hand, if the suspicious privacy disclosure is necessary to the benign code written by the developer, the developer can explicitly justify the necessary usage of the privacy disclosure in the app description.

D. Threat Model

We position AAPL as a suspicious privacy disclosure detection system, designed to uncover privacy leakage data flows in a given Android app. Instead of focusing on malicious privacy leakages that deliberately evade detection, AAPL targets efficient and scalable screening of a large number of apps, some of which may not respect user privacy and aggressively exploit user privacy in exchange of revenue. Malware detection is described in existing works [4], [21], [39], and is out of scope of this paper. Specifically, AAPL detects unobfuscated privacy disclosures that could not be justified by the general functionalities of the app. In addition, while AAPL should generally have a high accuracy, it does not guarantee a zero false positive rate or false negative rate as most other privacy disclosure detection systems [18], [26]. AAPL aims to have a low false positive rate (such that manual filtering is feasible) while also provide a large degree of code coverage (as demonstrated through evaluation).

III. AAPL Detection System

AAPL aims to address the problem of automatically differentiating privacy leaks from legitimate privacy disclosures with high accuracy. AAPL achieves this goal by proposing two major components: 1) Privacy Disclosure Analysis Module that detects all potential privacy disclosures in Android app, including legitimate privacy disclosures and privacy leaks, and 2)

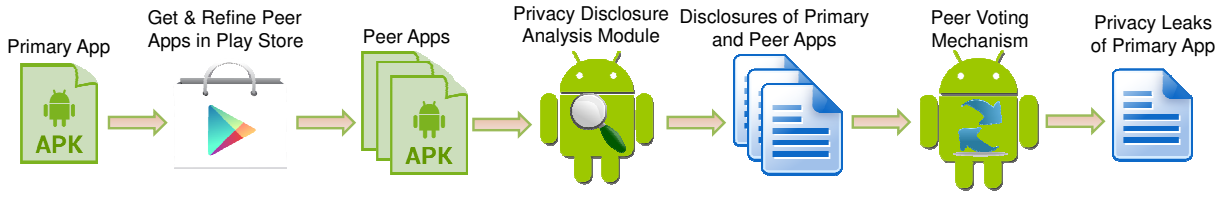

Fig. 2. Workflow of AAPL

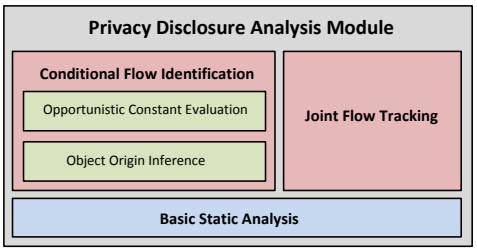

Fig. 3. Structure of privacy disclosure analysis module.

Privacy Leak Detection Module that differentiates privacy leaks from legitimate privacy disclosures using the novel peer voting mechanism. In this section, we first present the workflow of AAPL. Then, we describe the detailed design of the two components.

The workflow of AAPL is shown in Figure 2. Given a target app to be analyzed (*i.e.,* the primary app), we first identify the list of functionally similar peer apps based on the recommendations from the market providers (e.g., the Similar Apps list and Users Also Installed list from Google Play). Some noisy peer apps (that are not really functionally similar) are filtered by same-category policy and a NLP technique called semantic similarity [32]. After that, we build a highly accurate static privacy disclosure analysis module to detect all potential privacy disclosures in both primary app and its peer apps. We design several techniques that significantly improve the privacy disclosures detection accuracy for the programming paradigms of Android programs, such as opportunistic constant evaluation, object origin inference and joint flow tracking *etc.* Finally, we propose a novel peer app voting mechanism to automatically differentiate the legitimate privacy disclosure flows from the suspicious ones, which are likely to cause privacy leaks. AAPL does not depend on heavy domain knowledge or specific program execution context, and thus it greatly reduces the amount of human efforts.

A. Privacy Disclosure Analysis Module

The privacy disclosure analysis module of AAPL is responsible for generating privacy disclosure flow reports. Specifically, given an Android app, this module conducts highly accurate static program analysis to identify all potential privacy disclosure flows in this app, including both legitimate privacy disclosures and privacy leaks. The privacy disclosure analysis module is essentially a static taint tracking system. It detects all data-flows from the predefined sensitive data (*i.e., sources*) to the predefined channels (*i.e., sinks*). The privacy disclosure is modeled by data-flows connecting sources to sinks. In AAPL, we employ similar techniques from existing approaches [18], [26] for detecting entry points, building system dependence graph (SDG), and permutating and connecting the partial data-flows found by different entry points. However, they still suffer some limitations of handling conditional data-flows and joint data-flows, as illustrated in Figure 4. Correspondingly, we have designed and implemented a set of enhancements to further boost the accuracy of our detection module, including conditional flow identification through opportunistic constant evaluation, object origin inference, and joint flow tracking. as shown in Figure 3. With those improvements, we have achieved 31% more privacy disclosures detection rate while reducing the false positives by a factor of 5. In this section, we first show the motivations behind our improvements, and then elaborate how these improvements are designed to address limitations of existing static analysis systems for Android.

The first limitation of previous works [18], [26] exists in handling conditional sources/sinks[1]. The component-based programming paradigm of Android makes some of framework functions highly generic. The Android framework dictates a data model that revolves around the concepts of content *providers* and *resolvers*, in order to abstract shared OS objects (*e.g.,* files, sockets) and allow mediating (raw) access to them [2]. Every content provider needs to implement a predefined API that enables publishing and consuming data based on a simple URI addressing model (*i.e.,* using the content:// schema). The content resolver, on the other hand, is typically a global object that proxies data access requests to the appropriate content provider. Although such a scheme facilitates data-source *agnostic* apps by decoupling application logic from the underlying data layers, this design makes it difficult to track the *exact* data source accessed each request, as requests from more than a single data source are all proxied through a single content resolver. As illustrated in the 6th line in Figure 4, ContentResolver.query() is a very generic API in Android framework. All possible data related to content providers might use this API. We cannot simply define this API as a source/sink, since we cannot separate sensitive sources (*e.g.,* contacts, call logs) from non-sensitive data. Similarly, some generic objects such as IO streams might be used for different purposes, such as dealing with file IO objects or network sockets, as demonstrated in the 37th line in Figure 4. When define the sensitive sinks, it is hard to know to where the information is disclosed.

This abstraction paradigm makes the process of tracking sensitive data from sources to sinks difficult. If we blindly mark any invocation to conditional sources/sinks (*e.g.,* ContentResolver.query() and IO stream) as a

[1]Whether a generic API is a sensitive source/sink is dependent on the runtime values of its parameters or the type of the object it belongs to.

```
1  public void sendContactsToServer() throws IOException
2  {
3   Uri uri=ContactsContract.Contacts.CONTENT_URI;
4   /* Limitation: did not indicate if ``uri'' points to
5      sensitive data - requiring constant evaluation */
6   Cursor cur=getContentResolver().query(uri,null, null,
        null, null);
7   int contactsCounter = cur.getCount();
8   sendDataToServer(String.valueOf(contactsCounter));
9   if (contactsCounter == 0) return;
10  while (cur.moveToNext()) {
11    int idClnIdx = cur.getColumnIndex("_id");
12    String id = cur.getString(idClnIdx);
13    StringBuilder data = new StringBuilder(id);
14    int nameClnIdx = cur.getColumnIndex("display_name");
15    /* Limitation: did not track the joint flow from
16       ``rawFile'' to ``newFile'' - requiring joint flow
17       tracking */
18    String rawName = cur.getString(nameClnIdx);
19    File rawFile = new File(getFilesDir(), "filename");
20    writeDataToLocalFile(rawFile, rawName);
21    ...
22    File newFile = new File(getFilesDir(), "filename");
23    String newName = readDataFromLocalFile(newFile);
24    data.append(newName);
25    sendDataToServer(data.toString());
26  }
27 }
28
29 public void sendDataToServer(String Data) throws
       IOException
30 {
31  URL url=new URL("http://urlsample");
32  HttpsURLConnection conn=(HttpsURLConnection)
        url.openConnection();
33  OutputStream output=conn.getOutputStream();
34  /* Limitation: did not indicate if the generic
35     ``write()'' will write to sensitive sinks -
36     requiring object origin inference for ``output'' */
37  output.write(Data);
38 }
```

Fig. 4. Limitations with Previous Static Detection Systems

(sensitive) source/sink, we increase our detection rate at the expense of a high false positive rate (the same API call is used to access a plethora or different, and oftentimes insensitive data/channels [6]). On the other hand if we choose to skip them, then we will miss many extremely sensitive data/channels associated with conditional sources/sinks (*e.g.,* contacts, browse history, call log, SMS, and IO stream). To correctly identify conditional sources/sinks, we have to statically evaluate their possible values or types, which leads us to design `opportunistic constant evaluation` for conditional sources/sinks that are dependent on the values of their parameters in Section III-A1 and `object origin inference` for conditional sources/sinks that are dependent on the type of the object they belongs to in Section III-A2.

Another limitation of existing works [18], [26] is that existing taint flow tracking will stop when the data flow into locations outside the app code, such as databases, file systems, and other OS data structures. For example, the code from lines 18 to 25 in Figure 4 temporarily stores sensitive data in a non-sensitive local file, but later loads and sends the sensitive data out to remote server (*e.g.,* joint flow). We propose `joint flow tracking` in Section III-A3 to solve this issue.

1) Opportunistic Constant Evaluation: To identify the conditional sources/sinks that are dependent on

```
1  String str = "dead";
2  ...
3  switch (...) {
4    case 0:
5      str += "beef";
6      break;
7    case 1:
8      str += "caffe";
9      break;
10   ...
11   default:
12     str += "";
13 }
14 ...
15 System.out.println(str);
```

Fig. 5. Traditional constant folding example.

the values of their parameters (*e.g.,* the sensitivity of `ContentResolver.query()` depends on the value of its parameter `uri`), an intuitive solution is to statically collect a set of all possible values of the parameter in question and check if the set contains any interesting value that indicates a sensitive source/sink (*e.g.,* parameter `uri` in `ContentResolver.query()` points to `content://contacts`). We have devised a new technique, namely `opportunistic constant evaluation`, inspired by traditional constant folding technique. Constant folding (or constant propagation) is a standard compiler technique [9] that simplifies constant expressions at compile time. As an example, consider the following statement: `String str = "foo" + " " + "bar";`. Without the constant folding optimization this statement would resolve to a series of method invocations from the `String` class, *e.g.,* `String str = "foo".concat(" ".concat("bar"))`. If the compiler has constant folding enabled, it will directly initialize `str` with the (string) constant `"foo bar"`. Note that, such folding is performed only if it can be proven to be *safe*. For instance, in the snippet shown in Figure 5, the compiler will not constify the argument `str` of `println()`, since there are more than one data path that define `str` with a (different) value (the exact value of `str` depends on the `switch` condition, which in turn may depend on dynamic, run time computation). However, if the compiler determines that all cases apart "`default`" are dead code, it will replace `println(str)` with `println("dead")`, since `str` remains constant across all control and data paths leading to the `println()` statement. Similarly, value-flow analysis [7] can help find the variables that have same values as but different names with some constants.

`Opportunistic constant evaluation` is doing the folding in an opposite way. *Its main idea is to statically compute the set of all possible values of specific parameters along all program paths passed to the conditional sources/sinks, e.g., generic content resolvers, and decide the sensitivity of the source/sink based on the set.* As a result, in the example shown in Figure 5, it produces a set of all possible values `{deadbeef, deadcaffe, dead}`.

Let us consider again `ContentResolver.query()`. The actual content accessed with this call depends on the value of the first parameter (*i.e.,* `uri`) of `query()`. If `uri = "content://contacts/..."`, then contact information will be accessed. Likewise, if

uri = "content://mms-sms/...", MMS and SMS messages will be accessed. Assuming that both contact information and MMS/SMS content is sensitive, we should consider ContentResolver.query() as a sensitive data source in case uri is one of the above, but we should ignore it if, uri is used to access app-specific (insensitive) data. Although uri can be computed dynamically at run time, the Android framework provides a set of constant schemas that are typically used as templates by app authors when requesting access to content. For instance, Contacts.CONTENT_URI is a Uri object (declared as public static final) that uses the content schema to request access to contact information. Similarly, Browser.SEARCHES_URI is used for the browser search history. Our intuition is that in most cases, the uri parameter will either be constant at the point of use or constructed progressively (perhaps with the help of run time information) using URI objects from the Android framework (*i.e.,* start with a constant accessor, like Contacts.CONTENT_URI, and further specialize it by appending a variable suffix).

To facilitate conditional tracking, we first need to augment our list of predefined sources/sinks, with information regarding the parameter values of generic data and OS object accessors. We have manually analyzed the Android framework and identified about 600 cases that signify whether we should track a generic source/sink or not. For example, ContentResolver.query() is conditionally tracked in 48 cases, *e.g.,* when uri begins with the prefix content://call_log, content://mms, content://contacts, *etc.*

Armed with that information, we apply constant folding opportunistically in an inter-procedural manner, and collect as many constant values for the interesting parameters of conditional sources/sinks as possible. Notice that though traditional constant folding algorithms aim for safety (recall that in Figure 5 str will not be constified in the general case), we strive to be as *complete* as possible. Hence, for the code snippet in Figure 5, AAPL returns the following as possible values for str: {"dead", "deadbeef", "deadcaffe"}. To achieve that, leveraging system dependence graph (SDG) [24], we build a data-dependency only graph (DDG). We first compute a data-dependence sub-graph of the source/sink in question. Such a sub-graph is a DDG rooted with the conditional parameter(s) of the respective source/sink (taken from our augmented list). Every node of that sub-DDG represents a Static Single Assignment (SSA) statement related to the computation of the value of the parameter at the source/sink. We then perform a *post-order* DFS traversal and constify each node using abstract interpretation of the SSA statements. Note that the post-order traversal of the sub-DDG guarantees that before trying to constify a certain node, we will have already visited and constified all its predecessors (DDGs are directed graphs). The initial seeds to this process are SSA statements that correspond to leaf nodes in the sub-DDG graph and contain constants of primitive types (*e.g.,* strings, URIs, numerical values). Finally, once we have collected a set of possible constants for the respective parameter(s), we check every constant separately to see if it matches the value that indicates this is a source/sink.

2) Object Origin Inference: In Section III-A1, we show how we identify the conditional sources/sinks that depends on their parameter values. In this section, we present another kind of conditional sources/sinks whose conditions are the types of the objects they belong to. In the 37th line in Figure 4, the object output of generic type OutputStream can write data to either a local file or a remote server based on its channel. However, data-flow analysis itself can only reflect that the output is an object of type OutputStream. Without the knowledge of the concrete channel output is pointing to, we cannot figure out to where it will write data, and thus fail to determine whether output.write() is a sensitive sink. Type inference [3], [28] is a traditional technique used for analysis and optimization of object-oriented programs (*e.g.,* Java program). The basic idea of traditional type inference systems allocate a type variable with every slot and expression in the target program, then build a directed graph by performing the (top-down) data-flow propagations over all the type variables. Given that we are only interested in some particular conditional sources/sinks, we propose the lightweight and efficient object origin inference to infer the "derived type" of the interested object. Having the background with sub-DDG in constant evaluation, our origin inference approach is actually intuitive. Considering the same example, we first select the node of output in the SDG as a root node, and build its sub-DDG using backward slicing. Unlike constant evaluation, we now perform pre-order BFS traversal to find the closest node for constructing the object. We collected the API signatures of the constructors of all possible derived types. Upon encountering any constructor node (e.g., HttpsURLConnection.getOutputStream() in the given example) in the sub-DDG, we terminate the traversal and obtain the type information from the constructor node. Given the type information, we can confidently identify the sensitive source/sink confidently. In the given example, as output is a OutputStream object for HttpsURLConnection, we identify it as a sensitive sink. Similarly, we need to augment such conditional sources/sinks with sensitive type information. Including the ones we defined in Section III-A1, we modeled a set of augmented 66 source types and 15 sink types that we constructed semi-automatically using the API-to-permission mapping provided by PScout [6][2]

3) Joint Flow Tracking: In AAPL, the privacy disclosure detection problem is converted to the equivalent problem of checking for data-flows between sources and sinks. Naturally, the identification of sources/sinks becomes the prerequisite. Consider the example shown in the source code lines from 18 to 25 in Figure 4, in which the sensitive data rawName (a contact name) is first temporarily stored in the local file, and then loaded again to newName that is finally sent out to the remote server. From the perspective of privacy disclosure analysis, it can successfully identify cur.getString() as a source and output.write() in sendDataToServer() as a sink, and then check whether there is a flow between them. It turns out that the flow cannot be identified due to: 1) rawFile was not identified as a sensitive sink and newFile was not identified as a sensitive source; 2) the flow between rawFile and newFile is implicitly connected outside of app code, thus not identified. However, the sensitive data rawName is indeed sent to the remote server, as there is a joint flow from

[2]Note that while this set is adequate for testing and evaluation purposes (*i.e.,* it covers a relatively wide range of information leaks), it is by no means complete.

rawFile to output.write(). We propose using joint flow tracking to handle such issues. Unlike the mechanisms in opportunistic constant evaluation and fine-grained propagation policy, joint flow tracking records all potential sources/sinks even they point to non-sensitive resources/channels, and finds all sub-flows containing potential sources or potential sinks by conservatively matching all potential sinks with all potential sources. We can then join sub-flows together to complete privacy disclosure flows (if they exist). In the given example, even though rawFile is pointing to a non-sensitive local file, it is first recorded as a potential sink. Accordingly, newFile is recorded as a potential source. The sub-flow from rawName to rawFile and sub-flow from newFile to output.write() will be joined together to form a sensitive privacy disclosure flow. Note that the number joint flows is usually small (See Table II), the overhead of the iterative matching is minor.

B. Privacy Leak Detection Module

Our study shows most privacy disclosures (see Figure 7) are actually necessary to apps' core functionalities. Simply reporting all privacy disclosures to end users provides limited help, and the burden of consuming all privacy disclosures is extremely heavy. It is pressing to propose an automatic approach to differentiate the privacy leaks from legitimate privacy disclosures. We observed that identifying uncommon data-flows (*i.e.,* suspicious privacy disclosure flows) among peer apps with similar functionalities is an effective way to differentiate privacy leaks from legitimate privacy disclosures. In this section, we detail how we collect the peer apps for each primary app and leverage peer voting mechanism to find the privacy disclosure flows which are likely to cause privacy leak flows.

1) Peer Apps: Peer apps are defined from the perspective of users, which comprises the apps that are functionally similar to the primary app. In other words, a peer app could be an alternative app of the primary app to the users. AAPL aims to detect the privacy leaks of a given primary app via peer voting mechanism. Given the primary app, the first step is to derive its peer apps. There are several options to do this. For examples, we can simply use keyword-based search to collect peer apps, make use of machine learning techniques to classify apps based on some features (*e.g.,* category, description, permissions, and called APIs), or leverage Google Play recommendation system.

To evaluate the quality of keyword-based peer apps, we took the name of the primary app as the keyword to query Google Play, and manually verified the returned peer apps. Unfortunately, we found several obvious problems with this option: 1) If the given app is unpopular, the number of returned "peer apps" is usually very small (*e.g.,* less than 2), which is not enough for peer voting; 2) any app containing the keyword may appear in the returned apps list. For example, taking *Facebook* as a keyword, a game app Space Dog + Facebook Game is in the returned list, which is obviously not functionally similar to *Facebook*. Due to these fundamental issues with the keyword-based approach, we turn to think about the machine learning approach to collect peer apps. However, features, such as permission and API are not suitable since they would bias the peer apps to likely have similar privacy disclosures, defeating our original purpose of discovering uncommon privacy leaks.

We instead decide to adopt the existing Google Play recommendation system, which can provide a Similar Apps list (also called Users Also Viewed before), and a Users Also Installed list for each app. These two lists are derived from the users' experience. Google takes the user views and installation patterns and leverages on data mining to derive these two lists to help users find alternative apps. Even though the detail techniques behind the recommendation is a black box, these two lists indeed provide users meaningful choices in terms of selecting functionally similar apps.

It is possible these two lists may contain some noisy apps and polluting apps. There are two main kinds of noisy apps: accessory apps and the "most popular" apps. It can be expected that users who viewed/installed an app may also view/install its accessory app (*e.g.,* skin or photo downloading app), and the most popular apps (*e.g.,* browser app). Though providing accessory apps and "most popular" apps is a necessary feature for a recommendation system, AAPL prefers purer peer apps to produce more accurate results. Therefore, we seek for some way to filter out noisy apps. For "most popular" apps, the simple same-category policy can remove most of them. Only apps having same category as the primary app are selected as potential peer apps. To filter out other noisy apps, we observe NLP can help provide a ranking of similar apps based on the semantic similarity [32] of descriptions between them and primary app. For example, the description of facebook skin app only mentions its changing color functionality but none of facebook's core functionalities (*e.g.,* sharing photos and videos with friends, posts, text, chat and games). Such apps will be naturally put in a low ranking by NLP. With this intuition, we apply NLP on the similar app lists provided by Google Play, and rank them based on their semantic similarity with the primary app. Apps with low ranking will be excluded for peer apps, as shown in Section V-C1.

Similar apps recommended by Google Play that are published by the same publisher as primary app, are defined as polluting apps. In our evaluation, we indeed find many such cases that the developers release multiple "almost the same" apps, dramatically affecting the fairness of peer voting (Section III-B2) and resulting in false negative. To eliminate such polluting apps, we take the developer account into consideration, ensuring that the developer of every peer app is different from the one of the primary app.

The above filtering is not completely reliable. Fortunately, the peer voting mechanism does not require a perfect peer app list. As shown in Section V-C2, with the quality of current peer apps selection approach, we have already achieved good accuracy. Note that the peer voting mechanism is not bound to a specific peer app selection algorithm. If one can provide a better approach to produce peer app lists in the future, the final detection accuracy of AAPL will be further improved.

2) Peer Voting Mechanism: After detecting the privacy disclosures of primary app and its peer apps, the peer voting mechanism becomes intuitive. Given a particular privacy disclosure in each primary app, every (its) peer app has a chance to vote for it: every peer app needs to answer the question:

Do I have this privacy disclosure? If yes, the peer votes for 1; otherwise, vote for 0. The total number of votes with 1 is represented as `VotesNumber`, while the number of peer apps for the primary app of the given privacy disclosure is represented as `PeersNumber`. We can derive a new number, namely `privacy legitimacy`, which is calculated with the following formula:

$$\texttt{privacy legitimacy} = \texttt{VotesNumber/PeersNumber} \quad (1)$$

`privacy legitimacy` represents "how likely the privacy disclosure is legitimate in the primary app".

IV. Implementation

In this section, we detail our end-to-end implementation of the complete suspicious privacy disclosure flow detection system, including the privacy disclosure analysis module and the peer voting mechanism.

A. Privacy Disclosure Analysis Module

Similar to CHEX [26], our privacy disclosure flow analysis module is built on top of `Dalysis` [26] and IBM `WALA`. It takes as input an off-the-shelf apk file, and translates its Dalvik bytecode into an intermediate representation (IR), relying on `Dexlib`, a popular and open-sourced Dalvik bytecode parser for Android apps. After that, static single assignment (SSA) conversion is performed to leverage various built-in basic analyzers of `WALA`, *e.g.,* point-to analysis and the call graph building.

Given that existing works [1], [18], [26] have implemented similar static program analysis systems based on `WALA`, we skip the basic implementation details, and only elaborate on our core contributions described in Section III-A, which significantly improve the accuracy of privacy disclosure flow detection. These improvements account for about $6K$ SLoC in Java. In the following, we mainly focus on system dependency graph refinements conditional data-flow analysis. and post flow analysis.

1) Refining System Dependency Graph using Fine-grained Propagation Policy: Similar to the previous works [26], we do not include the framework code in our analysis scope, but choose to model the framework functions. This design choice helps us avoid significant overhead and inaccuracy caused by the use of reflection and complexity in framework code; on the other hand, it misses the flow propagation details inside the framework functions. To address this issue, when building the SDG using `WALA`, we first employ a default coarse-grained data propagation policy (*i.e.,* there are flows from parameters to return value, but not flows between parameters) for framework functions, and obtain a raw SDG. After that, we adopt manual modeling of commonly invoked framework functions at a fine granularity, and perform SDG refinement to insert or delete edges for the manually modelled framework functions.

Figure 6 shows an example of how we specify our fine-grained data propagation policy for a frequently used framework function. More specifically, each `propagationPolicy` tag contains a set of flow propagation policies of a specific framework method. Within each `propagationPolicy` tag, we could specify multiple data-flows between the parameters, return value, and the object

```
1  <PropagationPolicies>
2  <class name="java.lang.StringBuilder">
3  <method name="append">
4  <parameter name="this" type="java.lang.StringBuilder"
       />Next
5  <parameter name="str" type="java.lang.String" />
6  <return name="ret" type="java.lang.StringBuilder" />
7  <propagationPolicy from="str" to="this" />
8  <propagationPolicy from="str" to="ret" />
9  <propagationPolicy from="this" to="ret" />
10 </method>
11 ...
12 </class>
13 ...
14 </PropagationPolicies>
```

Fig. 6. Fine-grained Propagation Policy Example

instance. To make the coverage of fine-grained propagation policy as complete as possible, we collect all framework functions used in the app set used in evaluation V-B, and rank them based on their occurrences in descending order. We specify fine-grained policies as long as a framework function has 1) a flow from its parameter to parameter; or 2) no flow from any parameter to return value. Finally, our policy set covers 131 commonly used framework functions contained in 17 classes, including `String`, `Uri`, `StringBuilder`, `HttpUriRequest`, etc. In our in-depth evaluation in Section V-B, we did not find any false positive flow that is due to we have not specified fine-grained policies for the involved functions.

2) Implementing a Highly Accurate Detection Model: To accommodate the programming paradigm of Android, our primary goal is to accurately determine whether the conditional sources/sinks are indeed sensitive. We implemented the opportunistic constant evaluation and object origin inference components to support conditional source/sink as a back end analyzer.

For opportunistic constant evaluation, we first construct the sub-DDG for a statement in question (*e.g.,* for a generic data wrapper like `ContentProvider.query()`). Next, we traverse the sub-DDG using the technique outlined in Section III-A1. We extended the basic `WALA` statement node representation with additional fields for keeping sets of constant values. While performing the post-order DFS traversal each node is "constified" by considering the SSA instruction of the node, along with the constant sets of the predecessor nodes (*i.e.,* typically instruction operands). The opportunistic constant evaluation process is also guided by our manually specified "folding" policies (*e.g.,* we perform string concat for `StringBuilder.append()`), including special handling for all ALU instructions (*i.e.,* `ADD`, `AND`, `XOR`, `SHL`, `NEG`, *etc.*), as well as domain knowledge for all methods of primitive Java classes, such as `String`, `StringBuilder`, `Uri`, `Integer`, `Float`, `Boolean`, *etc.*

The implementation of object origin inference is similar, while it leverages the pre-order BFS traversal to find the closest constructor or assigner (*i.e.,* the origin) that can reflect the type information of the object in question (*e.g.,* `output` in 37th line in Figure 4). With the type information of the object, we can safely decide the sensitivities of the respective conditional sources/sinks. The design of joint flow tracking is described in Section III-A3; its implementation is straightforward, and

thus omitted here.

a) Post Flow Analysis: The discovered privacy disclosure flows may contain false positive ones that are mainly from data propagation. See the common example shown in lines 7 and 8 in Figure 4. `cur` is identified as a sensitive source, as `uri` points to `contacts`. And then `contactsCounter` derived from `cur` is sent to the remote server. This flow would be detected as a privacy disclosure. However, the value of `contactsCounter` is non-sensitive, which results in a false positive privacy disclosure. In order to reduce false positive, we have adopted a heuristic-based post flow analysis model to filter out different kinds of false positive flows: 1) only the non-sensitive metadata is disclosed (*e.g.,* only `contactsCounter` in above example is sent out); 2) sensitive data is processed by sensitivity-eliminating functions (*e.g.,* a hash function); 3) The sensitive data that is normally disclosed to particular sinks, *e.g.,* cookie is normally disclosed to remote server, should not be taken as a privacy leak. Note that, our heuristic-based may increase false negatives, and hard to be complete. However, based on the evaluation shown in Section V-B, `AAPL` has already achieved a good accuracy, in terms of a low false positive rate and a good detection rate.

B. Peer Voting Mechanism

To implement the peer voting mechanism, we first implement the noisy peer app filtering approach described in Section III-B1. Leveraging NLP to rank peer apps is the main component in the noisy peer app filtering process. There are about 5% of apps with non-english description. We skip them and leave the peer app list unchanged. More specifically, we adopt `NLTK` [3], a popular and easy-to-use Natural Language Toolkit for python program. Given descriptions of two apps, we preprocess them by lowercasing the texts and removing the punctuations inside them. We then tokenize the description texts and remove `English` stop words (*e.g.,* 'this', 'is', 'a', etc.). After that, we change the tokens into tf-idf [4] vectors. With tf-idf vectors for the given two apps, we can easily compute the cosine similarity [5] between them, which represents how similar the two descriptions are. In this way, we obtain similarities between each peer app and its primary app, from which we produce a ranked list of peer apps. Note that we need to further set a threshold of similarity to determine which ones are to be excluded (see Section V).

With the purified peer apps, we apply the privacy disclosure analysis on the primary app and their peer apps, and collect privacy disclosure flow results. We then conduct peer voting by counting the number of how many peer apps have the same privacy disclosure as the primary app. Finally, we derive the privacy legitimacy of each privacy disclosure in primary app by dividing the count by the number of peer apps.

V. Evaluation

We carry out systematic evaluations of `AAPL`, covering both the privacy disclosure detection system and the peer voting mechanism. In this section, we first describe our evaluation setup and methodology; then detail the evaluation results for each component. After that, we present some case studies of our detection results. Finally we discuss the source of inaccuracy and possible mitigation.

A. Setup and Methodology

We perform our experiments on a cluster of three servers, each equipped with a Xeon E5-1650 processor and 64 GB of memory. These servers are inter-linked with a 1Gbps Ethernet network link. With this setup, we are able to execute static analyses for 18 apps in parallel.

We collect a set of primary apps by running our privacy disclosure analysis on a set of most popular apps from the Google Play market, and discover apps that have privacy disclosure flows that may lead to privacy leaks. We then leverage the app market's recommendation system to derive the (refined) peer apps for each primary app. We again perform privacy disclosure analysis on the entire set of peer apps. Finally, we apply the analysis results in the peer voting mechanism to deduce the primary apps with potential of privacy leaks.

We evaluate the accuracy of both the privacy disclosure analysis and the peer voting mechanism using manual efforts. Due to the large quantity of results, we randomly select a subset of the results and investigate their correctness. For disclosure analysis, we perform disassembly and decompilation on the discovered disclosure code paths and examine their validity. For the peer voting mechanism, we present each primary app and its disclosure to 5 domain experts, who have been working on Android disclosure detection more than two years. They rate the legitimacy of the disclosure based on the following procedures:

1) Collect app semantics by reading app description, "What's New", screen-shots, and user reviews, and by installing and running the app;
2) Identify the functionalities the app potentially has;
3) Try to map the disclosures to the identified functionality – the successfully mapped disclosures are flagged legitimate.
4) Those disclosures could not map to any identified functionality are then flagged as privacy leaks with potential of privacy violations.

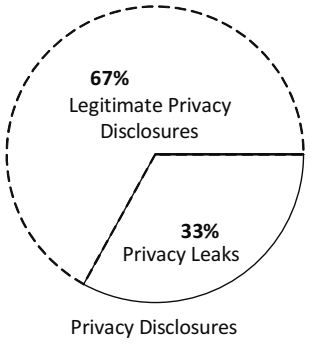

Fig. 7. Ratio of Legitimate Privacy Disclosure and Privacy Leaks

The ratings of the 5 experts on each app are merged as follows. A disclosure is considered as a legitimate disclosure

[3] http://www.nltk.org/

[4] tf-idf is short for term frequency-inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus

[5] Cosine similarity is a measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them

if 3 or more experts flag it as such; otherwise, it is a privacy leak. Finally, we label 532 unique privacy disclosures from 417 randomly chose primary apps. The results (see Figure 7) show there are 356 (67%) privacy disclosures are actually legitimate (*i.e.,* there are indeed necessary to apps' core functionalities). This confirms our observation that most privacy disclosures are legitimate. These labelled disclosures are then equally but randomly divided into "training set" and "testing set", each containing 178 privacy disclosures and 88 privacy leaks. Note that the training set is only used to determine the two parameters of `AAPL` (but not to learn a complicated machine learning model) to achieve the highest accuracy: 1) a parameter used as a similarity threshold to exclude noisy peer apps in the ranked similar app lists by NLP; 2) a parameter used as a privacy legitimacy threshold to differentiate legitimate disclosures from privacy leaks in the output of peer voting. Correspondingly, the testing set is used to verify effectiveness of `AAPL` with these two determined parameters. We take the accuracy in testing set as the one our current `AAPL` can achieve.

B. Disclosure Analysis

We conduct a large-scale experiment on the disclosure detection system. First, we analyze a set of $6,000$ most popular free apps and discover $2,777$ of them have privacy disclosures. Taking this $2,777$ as primary apps, we further calculate their peer apps. Overall, we analyze $40,456$ apps (including primary apps and peer apps), and find $18,100$ apps have privacy disclosures (*i.e.,* the overall detection rate is 44.7%). In the following, we first present the detailed analysis results, and then discuss the detection accuracy as well as the performance.

Table I presents a detailed classification of privacy disclosures discovered by the detection system. We select the top 10 most frequently disclosed privacy data, which accounts for over 97% of all disclosures, and show their corresponding percentage over all disclosures, total number of apps, as well as a breakdown by data sink types (*i.e.,* data is disclosed to network (Network Sink) or a local public location inside device (Local Sink)).

Privacy	Rate by Disclosures	Rate by Apps	Network Sink	Local Sink
Android ID	32.48%	21.38%	45.13%	54.87%
Location	26.15%	20.51%	33.56%	66.44%
Device ID	12.33%	11.22%	26.47%	73.53%
Contacts	8.08%	6.89%	4.49%	95.51%
External Storage	7.41%	6.73%	100.0%	0.0%
Phone Number	6.46%	6.07%	21.13%	78.87%
Browse History	1.72%	1.99%	0.0%	100.0%
Call Log	1.40%	1.12%	73.33%	26.67%
SMS	1.13%	1.02%	87.50%	12.50%
Cookie	0.41%	0.51%	0.0%	100.0%
Overall	97.57%	77.44%	37.40%	62.60%

TABLE I. DETAILED PRIVACY DISCLOSURE DETECTION RESULTS

We evaluate the detection accuracy by manually investigating 530 data-flows in 300 popular apps, and find a false positive rate of 6.7%. The majority of the false positives are related to over reduction of information along the data propagation path. For example, an app first retrieves user's contacts, then derives the size information from the contacts data, and finally sends out the size over the Internet. Because the size information contains too little entropy to recover meaningful data of the contacts, such information flow does not disclose privacy data. Although these kinds of information flows could be ruled out on a case-by-case basis, there is no general rule to effectively filter them[6]. As a result, we consider these detected information flows as false positive.

We further conduct an experiment to understand the effects of our static analysis improvements, as presented in section III-A. Table II presents a break-down of each improvement and its impact on the detection rate and accuracy. In this table, the `Detection Rate` shows how many apps have the privacy disclosures; the `Disclosures per App` is the average number of disclosures an app has. Note that the detection rate 48.4% on the last row of table II is slightly higher than the overall detection rate 44.7%, because this evaluation is performed on a smaller app set with the most popular apps. (*i.e.,* the chosen 300 popular apps rather than all 40.456 apps we analyzed).

In terms of performance, we find that the privacy disclosure flow analysis module has a throughput of 4.5 apps per minute in our three-machine cluster. And the analysis procedure can scale linearly with the available computation resources. However, there are 6.8% of the apps could not be analyzed under the pre-defined timeout threshold ($1,200$ seconds), and thus their analysis results are excluded from the evaluation. We find most of the timeout cases are due to a large number of entry points, which leads to exponential growth in inter-component permutations. We believe the high performance of privacy disclosure flow analysis module makes `AAPL` practical for screening a large-scale apps.

C. Peer Voting Mechanism Evaluation

To illustrate the effectiveness of peer voting, we use the following standard metrics:

- `True positives (TP)`: a privacy leak that `AAPL` correctly detects as a privacy leak
- `False positives (FP)`: a legitimate disclosure that `AAPL` incorrectly detects as a privacy leak
- `True negatives (TN)`: a legitimate disclosure that `AAPL` correctly detects as a legitimate disclosure
- `False negatives (FN)`: a privacy leak that `AAPL` incorrectly detects as a legitimate disclosure
- `False positive rate (FPR) = FP/(TN + FP)`
- `False negative rate (FNR) = FN/(TP + FN)`
- `Accuracy = (TP + TN)/(TP + FN + TN + FP)`

We evaluate the effectiveness and correctness of the peer voting mechanism using the "ground-truth" data set obtained via manual investigations, as described in Section V-A. The expert labeled data set is randomly divided into a training set and a testing set. Note that, our evaluation excludes "phone-state" related disclosure (*e.g.,* Device ID and Android ID). The reason is that phone-state data is not functionality-related,

[6]For example, instead of the size information, the app may derive a numeric representation of the contact's phone number, in which case the data-flow does still carry sensitive information.

Improvements	Detection Rate	Disclosures per App	False Positive Rate
Before Improvements	36.9%	0.82	34.2%
+Fine-grained Propagation Policy	40.3% (9.0%↑)	1.00 (22.3%↑)	28.1%
+Constant Evaluation	49.3% (33.4%↑)	1.72 (109.0%↑)	16.5%
+Object Origin Inference	52.5% (42.1%↑)	1.76 (114.6%↑)	16.1%
+Joint Flow Tracking	52.8% (43.0%↑)	1.77 (115.1%↑)	16.1%
+Post Flow Analysis (With All Improvements)	48.4% (31.0%↑)	1.40 (70.8%↑)	6.7%

TABLE II. EVALUATION OF ACCUMULATIVE IMPROVEMENTS

as any kind of app may use phone-state data for different purposes. The average number of the original similar apps provided by Google Play is about 70.

1) Parameters of AAPL*:* As mentioned in Section V-A, in order to achieve the highest accuracy, there are two parameters to be selected based on the training set: the similarity threshold used for choosing peer apps from the ranked similar apps, and the legitimacy (*i.e.,* percentage of peer app having same disclosures as primary app) threshold used for detecting privacy leaks. Given this simple detection model, determining these parameters is intuitive: we exhaustively iterate each possible value for both parameters and check which pair of values can produce the best accuracy. Each similar app is bound with a semantic similarity score ranges from 0.000 to 1.000 , and the legitimacy score ranges from 1.0% to 100.0%. After the iteration, we observe a limitation when we use similarity score as the first parameter to select peer apps: similarities in different similar app lists are extremely diverse due to the various lengths of descriptions in primary apps. More 10% primary apps do not have any similar app with similarity greater than the average similarity (through the whole app set) - 0.148. For example, FriendPaper Live Wallpaper has 79 similar apps, most of which are also wallpaper-related apps (and thus truly similar apps). However, due to its description only contains one sentence and several words, even its most similar apps have less than 0.05 similarities with it. On the other hand when the description of a primary app has a moderate length and can reflect app's functionalities properly, most of its similar apps may have a high similarity (greater than 0.5). In other words, the computed similarity score cannot be treated universally, but is specific to the primary app. Considering each list has different number of similar apps, we alternatively choose to use ratio (*i.e.,* the percentage of top similar app to be selected) as a parameter to select top similar apps as peer apps, which ranges from 1% to 100%. Figure 8 shows the best accuracy AAPL can achieve when we set the different ratios. More specifically, when selecting top 78% (ranked) similar app as peer apps, we achieve the best accuracy at 90.2% with false positive rate 11.3% and false negative rate 6.8%, based on training set. The privacy legitimacy parameter is not drawn in this figure due to limited space, which is usually around 2%. In the point AAPL achieves the best accuracy, the privacy legitimacy parameter is 1.8%.

2) Accuracy of AAPL*:* After the parameters of AAPL are determined, we apply AAPL to the testing set to evaluate the actual accuracy AAPL can achieve. The results, derived from testing set, show AAPL can achieve a high accuracy. More specifically, AAPL achieves an accuracy 88.7% with false positive rate 10.7% and false negative rate 12.5% . We additionally apply AAPL to some specific disclosures. For certain categories of disclosures, such as contacts, AAPL

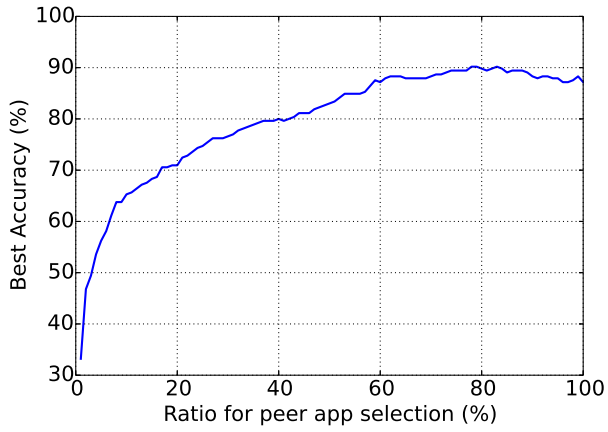

Fig. 8. Best Accuracy for Different Ratio of Peer Apps

achieves an even higher accuracy of 94.6% with false positive rate 4.4% and false negative rate 9.9% .

3) Comparison to NLP-based Peer App Selection: One interesting question is how good will the accuracy be if we only use NLP to select peer app. We also perform evaluation on NLP-based (only) peer app selection, using the same methodology (*i.e.,* determining parameters of AAPL based on the same training set), and computing accuracy based on the same testing set. Note that a limitation of NLP-based approach is that it is hard to collect a complete app set, unless we gain Google's support. Instead of collecting a complete app set, we take all collected app in our evaluation (*i.e.,* 40,456 apps), as the app set. Similarly, given a primary app, we apply NLP to calculating semantic similarity between each app in the app set and it. Then we rank all apps based on their similarity scores. After that, we again iteratively choose number (from 1 to 200) of top ranked apps and privacy legitimacy score (from 0.0% to 100.0%) as the two parameters of AAPL. After running AAPL on the training set and testing set, we find the best accuracy 81.3% with false positive rate 15.2% and false negative rate 25.4% is achieved when choosing top 94 ranked similar apps as peer apps and 1.9% privacy legitimacy to determine the legitimacy of disclosures. This result indicates that combining Google Play recommendation system and NLP for peer app selection can help AAPL achieve better accuracy than only using NLP, based on our current training and testing data set. This is probably because of a fundamental limitation: many apps do not provide sufficient or accurate descriptions for their core functionalities.

4) Comparison to Permission-based Detection: We further evaluate the contribution of the disclosure detection to the accuracy of the peer voting results by applying the same

peer voting technique based on permission uses instead of privacy disclosure flows. Intuitively, each privacy disclosure of the primary app corresponds to a sensitive information source and a public or network data sink. And almost all sensitive information source in the Android system is protected by certain permission. As a result, we could simply map a particular privacy disclosure in the primary app to the use of the permission that protects the corresponding sensitive information source. The peer voting is thus defined as the popularity of using the permission that protects particular privacy information.

We evaluate the permission use based peer voting result against the expert tagged "ground-truth" data set. As shown in Figure 9, compared to AAPL detection results, permission use based peer voting yields significantly worst results than AAPL, in terms of both precision and recall. Therefore, it is evident that the high quality privacy disclosure flow detection results are essential to achieve good peer voting results.

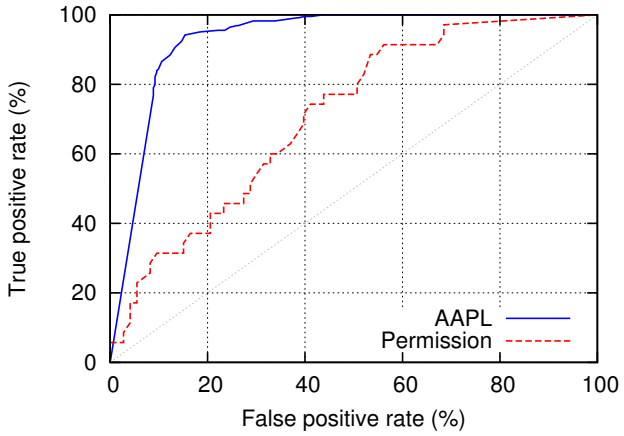

Fig. 9. ROCs for AAPL and Permission-based Detection

5) Case Studies and Sources of FP & FN: Our manual verification has revealed many privacy leak cases, as well as false positive & false negative cases. Table III shows some notable cases, which we will briefly explain in this section.

The app in case 1 is a keyboard input app, which is used to type/write both English and Chinese. We discover that it has a privacy disclosure flows that sends contact information to an URL address. Because no other keyboard apps have such a privacy disclosure flow, AAPL correctly flagged it out as a privacy leak.

The apps in case 2 and 3 disclose sensitive data (*e.g.,* authtoken and cookies), to a log file, which might be read by other apps with log reading permissions[7] Such privacy disclosure flows enable attackers to abuse the credentials stolen from the log file. AAPL also automatically flagged them out as a privacy leak.

We have found multiple cases where some sensitive data is leaked for certain purpose that could also be achieve with less sensitive data. In case 4, phone number is leaked as unique client ID. However, for the same purpose, the app could use the AndroidID instead. Thanks to peer voting, AAPL identifies these apps which are less privacy-legitimacy than their peers.

During our evaluation, we discover that many apps that contain privacy leaks have been removed from the Google Play market. Case 5 falls into such category. In addition, we have find that third-party library introduced privacy leak are prevalent. For example, in case 6, the sensitive data-flow is detected inside their embedded Ad library.

Our manual verification also helps us to understand the causes for false positives and false negatives in the peer voting mechanism. For false positives, the two major causes are: 1) The size of peer apps is not sufficient. After the polluting apps filtering, some clusters become very small. In case 7, com.easyandroid.free.mms only has 5 peer apps after the polluting app filtering, which results in less stable voting results, and thus the false positive. 2) The primary app has some special functionalities. In case 8, the contact printing functionality in the cloud-printing app is special, as its peer apps are generally designed for printing files but not contacts. As a result, although the disclosure is indeed necessary and expected to the users, AAPL still flags it out as "abnormal".

For false negatives, there are also two major causes:1) The privacy leaks inside popular third-party libraries result in similar effect of the polluting apps described in Section III-B1. In case 9, phone number disclosure is found in the ad library, but is not flagged out by AAPL. 2) Prevalent privacy indiscretion practices. Because AAPL makes decision based on voting, it is not capable of detection privacy leaks if the majority of apps exhibit similar behaviors. In case 10, AAPL does not flag the disclosure which writes cookies to log file, unlike case 2.

D. Another AAPL Application

Leveraging AAPL's capability of privacy disclosure discovery, we extend the idea of peer app comparison to another scenario—evaluating the privacy disclosure between official and unofficial apps. We define an unofficial app as an app from alternative Android markets with the same name, icon, and advertised functionalities as the corresponding app from Google Play. In other words, unofficial apps appear indistinguishable from the official ones to a normal user; however, because the alternative markets usually lack quality control and security assurance, unofficial apps are more likely to abuse users' privacy.

We download the top 150 free apps from Google Play, and their unofficial counterparts from Wandoujia [8], one of largest third-party mobile app distribution platforms, which collects apps from different major third-party Android markets. We first process all apps using AAPL and uncover each app's privacy disclosures. We find that while most unofficial apps have identical privacy disclosures to their respective official apps, there are 30 unofficial apps exhibit different privacy disclosures. We manually analyze these 30 unofficial apps and discover that 23 apps have more privacy disclosures, and 7 apps have less privacy disclosures than their corresponding official apps, respectively.

[7] Although the logging system of Android has been revised since version 4.1 so that each app uses independent log targets, currently there are still 37.5% of devices running older version Android systems, according to Google.

[8] http://www.wandoujia.com/

Case #	Type	App	Detected Leak	# of Peer Apps	Privacy Legitimacy
1	True Positive	$com.linpusime_tc.android.linpus_tckbd$	Contacts -> Url	20	0%
2	True Positive	$com.apptivateme.next.hrdp$	Cookie -> Log	15	0%
3	True Positive	$com.mobilecuriosity.ilovecooking$	Account->Log	14	0%
4	True Positive	$simosoftprojects.musicplayerforpad$	Phone Number -> Url	21	0%
5	True Positive	$com.webwag.alertmachine$	MMS -> Log	18	0%
6	True Positive	$com.scan.master$	Location -> Http	18	0%
7	False Positive	$com.easyandroid.free.mms$	PhoneNumber -> Http	5	0%
8	False Positive	$com.paulosl f.cloudprint$	Contacts -> Http	13	0%
9	False Negative	$com.blue.batterywidgetLedAndroid$	Phone Number -> Http	11	25.0%
10	False Negative	$app.angeldroid.safenotepad$	Cookie -> Log	17	13.3%

TABLE III. CASE STUDIES

Among the 23 apps with more privacy disclosures, we find three reasons for the increased leakages: 1) Modified control flow: the unofficial apps have modified components with potentially malicious code injected, which introduced more privacy leaks. 2) Replaced library: the unofficial apps have bundled different ad libraries, which contain more privacy leaks. 3) Cross component privacy leak: some additional privacy disclosures are found with source inside the app's own component but sink inside the ad library, and vice versa. We think this finding is quite interesting, and it may represent a new kind of leak, where the ad library may be exploited in participating in privacy leaks. Such a privacy leak behavior was never reported before in previous research in ad library related privacy leaks, such as [33]. Among the 7 apps with less privacy disclosures, we find two reasons for the reduced leakage: 1) Earlier versions: the unofficial apps are modified based on earlier versions of the official ones, which had less privacy leaks. 2) Replaced library: the unofficial apps have bundled different ad libraries, which contain less privacy leaks.

Based on the analysis of the chosen apps, we find more than 15% unofficial apps exhibit more privacy disclosures than official ones, most of which are repackaged to contain modified ad libraries. We recommend users download apps from Google Play to better protect privacy.

VI. LIMITATIONS

There are several limitations with AAPL. As other static analysis systems [1], [26] on Android, AAPL cannot detect the disclosures caused by Java reflection, code encryption, or dynamical code loading. As mentioned in Section III-B2, polluting attack may also bypass AAPL detection. Currently, we proposed a lightweight filtering approach using developer account comparison. Naturally, if the developer adopts multiples different accounts, this filtering might be bypassed. However, this will dramatically increase the cost for attacker to publish polluting apps in this way.

VII. RELATED WORK

There are a plethora of research efforts [12]–[14], [18], [20], [22], [26], [31] on detecting, measuring and understanding privacy disclosures on different mobile platforms. In terms of techniques of detecting flows of privacy disclosures inside an app, there are two major approaches. Dynamic taint analysis [13] sees the truth about what had happened. The challenges are either deployment on the phones with low overhead, or dynamic testing with good code coverage. TaintDroid [13] is one of the pioneer works using dynamic taint analysis to detect privacy disclosures. AppsPlayground [31] employs TaintDroid [13] and automated software testing technique to detect privacy disclosures. Due to low code coverage issue and relatively high overhead, this technology has not been well adopted yet. Static analysis sees the future (what could happen) with good coverage, and is more scalable, but it remains a challenge to ensure a low false positive rate. Many automated systems on app privacy disclosure screening are based on static analysis, such as PiOS [12], Woodpecker [20], CHEX [26] and AndroidLeaks [18]. Accurately detecting privacy disclosures statically needs to models Android specific programming paradigms. Woodpecker [20] and AndroidLeaks [18] both consider implicit control transfers such as threads and callbacks. CHEX [26] propose techniques to discover entry points inside Android apps and leverage entry point permutation techniques to detect the information flows related to privacy disclosures. FlowDroid [5] improves the precision of static taint analysis on Android apps by solving challenges of static analysis: handling the Android-specific lifecycle and the system event callback, identifying sources from user interaction fields and resolving alias in Java code.

AAPL not only takes the advantage of these previous wisdoms, but also statically solves the conditional sources/sinks by building the inter-procedural constant evaluation and concrete object type inference. Moreover, we also chain non-sensitive sub-flows to form sensitive data-flow using joint flow tracking.

All the aforementioned automation techniques do not try to analyze or to understand whether the detected privacy disclosures has any relationship with the app's functionality or user expectation. Instead, the security experts or the users have to understand such relationship manually. AppFence [23] provides fake data or information flow blocking for privacy control, but it remains challenging to automatically figure out whether those policies will affect the functionalities of the apps. Nadkarni *et al.*, propose a nice runtime enforcement and policy for proper information disclosure cross apps [27], in which, the policies are specified by experts or users, which is complementary to the approaches try to detect privacy disclosures in the apps statically. Appintent [35] employs static analysis and symbolic execution to understand how user actions trigger the information flow. It also provides program execution context alongside the detected privacy disclosure, which may help experts diagnose detected privacy disclosure. WHYPER [29] leverages a nature language processing technique to understand whether the application description reflect the permission usage. AAPL takes a novel complementary approach—evaluating the legitimacy of privacy disclosure based on the "criteria privacy disclosures" extracted from peer

apps.

CHABADA [19] applies topic modelling, an NLP technology to detecting malicious behaviors of Android apps. It generates clusters according to the topic, which consists of a cluster of words that frequently occur together. Then, it tries to detect the outliers as malicious behaviors. CHABADA cannot precisely find out privacy leaks since it just identifies the sensitive APIs without tracing the data-flow between the source and sink sensitive APIs. Our approach identifies suspicious privacy leaks and helps the users choose more "conservative" applications. Although previous works have employ app similarity and clustering for re-packing detection [37], we leverage on app recommendation system to find peer apps to compare their privacy disclosures.

There are many research efforts on Android related security problems. Enck *et al.*are the pioneers on Android permission policy study [15]. Further Felt *et al.* [16] and Au *et al.* [6] map the Android permissions to APIs, and study the permission usages. Peng *et al.* [30] and Chakradeo *et al.* [10] leverage machine learning approaches to predict the potential risk of an app with a specific permission set. A number of studies [8], [11], [17], [20], [26] focus on confused deputy attack surface for Android platform and apps, and Android malware analysis [36], [37], [39], [41]. Recently, several vulnerabilities and attacks have been found in in low-level Android modified Linux [25], [38] and Android's eco-system [34].

VIII. Conclusion

In this paper, we present the design and implementation of the AAPL system, which detects real privacy leaks in Android apps that negatively impact end-users. AAPL conducts specialized static analysis and achieves significant improvements over previous work in terms of coverage, efficiency, and accuracy. By comparing detected privacy disclosures in primary apps with those of the peers, AAPL effectively rules out the common and legitimate disclosures, exposing only those privacy leaks that cannot be associated with apps' functionalities. The results show that our peer voting-based approach can successfully remove the overwhelming noise (*i.e.,* false alerts on legitimate disclosures) from which most similar detection systems suffer. The evaluation demonstrates that AAPL scores a high accuracy of 88.7% with a 10.7% false positive rate and a 12.5% false negative rate, at a high throughput (4.5 apps per minutes on a three-machine cluster). As a result, AAPL can greatly increase the detection rate of threatening privacy leaks, and at the same time, considerably reduce the manually efforts required from security analysts or end-users.

Acknowledgements

The authors would like to thank the anonymous reviewers their helpful feedback, as well as our operations staff for their proofreading efforts. Wenke Lee, Kangjie Lu, and Cong Zheng were supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, and CNS-1149051, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

References

[1] "Scandroid: Automated security certification of android applications," http://www.cs.umd.edu/avik/projects/scandroidascaa.

[2] "Android application components," http://developer.android.com/guide/topics/fundamentals.html#Components, April 2012. [Online]. Available: http://developer.android.com/guide/topics/fundamentals.html#Components

[3] O. Agesen, "The cartesian product algorithm: Simple and precise type inference of parametric polymorphism," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, ser. ECOOP '95. London, UK, UK: Springer-Verlag, 1995, pp. 2–26. [Online]. Available: http://dl.acm.org/citation.cfm?id=646153.679533

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proceedings of the 21th Network and Distributed System Security Symposium (NDSS)*, 2014.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382222

[7] R. Bodík and S. Anik, "Path-sensitive value-flow analysis," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98. New York, NY, USA: ACM, 1998, pp. 237–251. [Online]. Available: http://doi.acm.org/10.1145/268946.268966

[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[9] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '86. New York, NY, USA: ACM, 1986, pp. 152–161. [Online]. Available: http://doi.acm.org/10.1145/12276.13327

[10] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: Triage for market-scale mobile malware analysis," in *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '13. New York, NY, USA: ACM, 2013, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2462096.2462100

[11] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proceedings of the 20th USENIX conference on Security*, San Francisco, CA, Aug. 2011.

[12] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*, 2011.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*. Berkeley, CA, USA: USENIX Association,

2011, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028088

[15] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653691

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[17] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: attacks and defenses," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028089

[18] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, ser. TRUST'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 291–307. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30921-2_17

[19] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1025–1035. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568276

[20] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[21] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 281–294. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307663

[22] J. Han, Q. Yan, D. Gao, J. Zhou, and R. Deng, "Comparing mobile privacy protection through cross-platform applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[23] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 639–652. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046780

[24] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *SIGPLAN Not.*, vol. 23, no. 7, pp. 35–46, Jun. 1988. [Online]. Available: http://doi.acm.org/10.1145/960116.53994

[25] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened aslr on android," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 424–439. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.34

[26] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382223

[27] A. Nadkarni and W. Enck, "Preventing accidental data disclosure in modern operating systems," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1029–1042. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516677

[28] J. Palsberg and M. I. Schwartzbach, "Object-oriented type inference," in *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '91. New York, NY, USA: ACM, 1991, pp. 146–161. [Online]. Available: http://doi.acm.org/10.1145/117954.117965

[29] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 527–542. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/pandita

[30] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 241–252.

[31] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 209–220. [Online]. Available: http://doi.acm.org/10.1145/2435349.2435379

[32] P. Resnik, "Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language," 1999.

[33] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *Workshop on Mobile Security Technologies (MoST)*. Citeseer, 2012.

[34] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your android, elevating my malware: Privilege escalation through mobile os updating," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 393–408. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.32

[35] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1043–1054. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516676

[36] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381950

[37] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: ACM, 2012, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133640

[38] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 409–423. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.33

[39] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.16

[40] ——, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[41] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.