Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps

Yuhong Nan*, Zhemin Yang*‡, Xiaofeng Wang†, Yuan Zhang*, Donglai Zhu* and Min Yang*§
*School of Computer Science, Fudan University
‡Shanghai Insitute of Intelligent Electroics & Systems
§Shanghai Institute for Advanced Communication and Data Science
Shanghai Key Laboratory of Data Science
†Indiana University Bloomington
{nanyuhong, yangzhemin, yuanxzhang, zhudl, m_yang}@fudan.edu.cn, xw7@indiana.edu

***Abstract*—A long-standing challenge in analyzing information leaks within mobile apps is to automatically identify the code operating on sensitive data. With all existing solutions relying on System APIs (e.g., IMEI, GPS location) or features of user interfaces (UI), the content from app servers, like user's Facebook profile, payment history, fall through the crack. Finding such content is important given the fact that most apps today are web applications, whose critical data are often on the server side. In the meantime, operations on the data within mobile apps are often hard to capture, since all server-side information is delivered to the app in the same way, sensitive or not.**

A unique observation of our research is that in modern apps, a program is essentially a semantics-rich documentation carrying meaningful program elements such as method names, variables and constants that reveal the sensitive data involved, even when the program is under moderate obfuscation. Leveraging this observation, we develop a novel semantics-driven solution for automatic discovery of sensitive user data, including those from the server side. Our approach utilizes natural language processing (NLP) to automatically locate the program elements (variables, methods, etc.) of interest, and then performs a learning-based program structure analysis to accurately identify those indeed carrying sensitive content. Using this new technique, we analyzed 445,668 popular apps, an unprecedented scale for this type of research. Our work brings to light the pervasiveness of information leaks, and the channels through which the leaks happen, including unintentional over-sharing across libraries and aggressive data acquisition behaviors. Further we found that many high-profile apps and libraries are involved in such leaks. Our findings contribute to a better understanding of the privacy risk in mobile apps and also highlight the importance of data protection in today's software composition.

I. Introduction

Mobile apps today are more composed than written, often built on top of existing web services (e.g., analytics or single-sign-on SDK). Such functionality composition, however, comes with significant privacy implications: private user information given to an app could be further shared to other parties through their components integrated within the app (e.g., libraries), in the absence of the user's consent. Indeed, prior research reveals that third-party services like ad libraries and analytics aggressively collect sensitive device information (e.g, IMEI, phone number, and GPS location data) [22], [37], [40]. Less noticeable here is the disclosure of the private user data an app downloads from its cloud or uploads from its local file, which could become completely oblivious to the user.

As an example, Figure 1 illustrates how *The-Paper* [14], one of the most popular Chinese news apps, works. The app integrates a third-party library *ShareSDK* [12] for sharing news posts to Weibo, a popular Chinese social-media platform, through its APIs. A problem we found is that the library actually acquires the user's access token, without a proper authorization, from Weibo and further utilizes it to gather the user's personal information (like one's detail profiles, her social activities, etc.) from the Weibo cloud. Unlike access to on-device data, which requires permissions from the user, or manually entering secrets (e.g., password) into the app's UI, collecting such server-side information is completely unaware to the user, since there is no user involvements (e.g., permission granting) at all before the information is exposed to *ShareSDK* and delivered to the untrusted party.

Such information disclosure is serious and can also be pervasive, given the fact that most mobile apps are essentially web applications, keeping most of their sensitive user data on the server side. An in-depth study to understand the scope and magnitude of the problem at a large scale, however, has never been done before, due to the technical challenge in automatic identification of such data sources inside the app code.

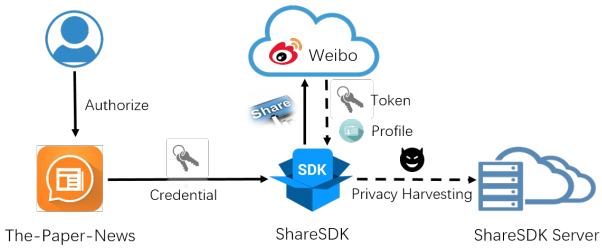

Fig. 1. User's sensitive data in Weibo server leaks to another service without her consent

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23092
www.ndss-symposium.org

Leakage analysis: challenges. More specifically, to find information leaks in an app, first one needs to locate the sources of sensitive data within the app code. Typically, these sources are discovered from the program based upon a set of System APIs that handle private on-device data, such as IMEI, phone number, GPS locations, etc. However, as mentioned earlier, private information comes from various sources, which can hardly be covered by these manually labeled System APIs. An example is user interfaces (UIs), whose inputs can be sensitive (e.g., password, home address) or public (e.g., comments) from the same API (e.g., *editText.getText()*). They are classified in the prior research [25], [32] using the semantics of their context, particularly tags of GUI items (such as the string "Password" right in front of a password entry). More complicated here is the user information managed by the app, which can be stored in local files or the app's server-side database. Loading such information into the app goes through generic APIs without any tags (file access, network communication), thereby giving little clue about the importance of the data transferred. As a result, disclosure of such information to unauthorized parties cannot be easily discovered.

```
1  # Getting location data in somewhere
2  Location location =
       LocationManager.getLastKnownLocation();
3  this.locationStr =
4    "latitude"+ location.getLatitude() + "\n"
5    + "longitude" + location.getLongitude();
6   ....
7  # Gathering user profile in somewhere else and
       send to server
8  # Method getUserBasicInfo()
9  Json fBUserJson = getDataFromFacebook();
10 ...
11 HashMap basicInfo = new HashMap<String, String>();
12 basicInfo.put("first_name",
       fBUserJson.get("First_name"));
13 basicInfo.put("last_name",
       fBUserJson.get("Last_name"));
14 basicInfo.put("last_location", this.locationStr);
15 ...
16 return basicInfo;
```

Fig. 2. Motivating example. Code snips from app *SnapTee* in Google-Play

A key observation in our research is that most apps today contain a large amount of semantic information for supporting their development and maintenance. As an example, we can see from the code snippet of a real-world app *SnapTee* [13] in Figure 2 that variables, functions, methods and other program elements are all given meaningful names, and plain-text content (strings in double quotation marks) is included in the code to explain other related content such as the value of a specific key. Further, these program elements tend to be organized in distinctive ways within the app, supporting unique operations on sensitive user data: for example, formating the information as key-value pairs and storing them in a HashMap (line 12-16 in Figure 2). Essentially, the whole program here can be viewed as a semantics-rich dataset, from which sensitive user content can be discovered with proper data analysis techniques. Such semantic information could also help information-flow tracking (which often cannot be done both efficiently and accurately), through connecting program locations to related semantics (e.g., directly confirming the presence of location data at line 14 from the constant "last_location", instead of tracking the data flow from the geolocation API at line 4).

Semantic clue discovery. Based upon the observation, we developed a new technique that automatically mines app code to recover semantic "clues" indicating the presence of sensitive information, which enables an effective leakage analysis across a large number of popular apps (Section V). Our technique, called *ClueFinder*, first utilizes a set of keywords, prefixes and unique acronyms representing various types of sensitive user information to identify the program elements (methods, variables, constants, etc.) that might involve sensitive content (e.g., *getUserPwd*, *home_addr*, "Last_name"). These elements are then inspected through Natural Language Processing (NLP), to remove those not representing any sensitive content. Oftentimes, variables, constants and method names carrying privacy-related terms end up being unrelated to sensitive information. For example, the method *getStreetViewActivity* includes the address-related keyword "street" but clearly does not involve private data. Another example is the constant *"invalid input for home directory"*, which has nothing to do with the user's home. To identify these false positive instances, ClueFinder performs a grammatical analysis, finding the matched terms or prefixes or acronyms not serving as the "theme" of their semantic context: for example, the word "street" here only plays the role of describing "activity", which is the true subject of the whole term (the activity name). On the other hand, when a keyword acting as a noun in its element and also as a subject of a verb (e.g., "getEmail"), it looks more like a clue for the presence of operations on sensitive user data.

Learning-Based identification. Such semantics analysis alone, however, can still be *insufficient* to avoid false positives, that is, mistakenly reporting a non-sensitive program element as involving sensitive content: e.g., sending a message with a constant-string *setMessage("are you sure to delete account?")* or throwing an exception like *formatInvalidExp("username", Exception e)*. To address this issue, ClueFinder further evaluates the program structures related to those identified elements, looking for the operations most likely to happen on sensitive user data. More specifically, it runs a machine-learning approach to classify the program statements containing such elements, based upon a set of key program structural features (Section III-C). For example, in Figure 2, line 14, we expect that within a method invocation statement *basicInfo.put()*, an identified constant text string involving sensitive keywords ("location") appears together with a variable parameter of a data type (String for the variable "locationStr"), which likely indicates the presence of a key-value pair. Note that this feature helps exclude the operation that simply displays the text with keywords (e.g., "account"), as in the aforementioned example *"are you sure to delete account?"*. Altogether we identified 5 features and trained an SVM model based upon the features to discover sensitive-data related operations from Android code, thus to identify the actual private content in mobile apps.

The design of ClueFinder enables efficient discovery of sensitive data sources, covering not only those labeled by System APIs, but also server-side private data (e.g., user profiles) and other content controlled by individual apps. Even in the presence of moderate obfuscation (e.g., produced by ProGuard [9]), our semantics-based approach still works, thanks to

the program features that needs to be preserved during obfuscation to avoid disrupting an app's normal execution (e.g., API names, parameters, constants, even some data operations, see Section IV-B). Although ClueFinder is primarily designed to find hidden data sources, we show that the semantic knowledge recovered by our approach also supports a more efficient data-flow tracking (see Section III-C), which enables a large-scale leakage analysis.

We implemented ClueFinder and evaluated its effectiveness in our research (Section IV). The experimental results show that ClueFinder accurately discovers sensitive data sources in app code (with a precision of 91.5%), significantly outperforming all prior approaches [35], [25], [32], [26], in terms of both coverage and precision.

Measurement and findings. Armed with more sensitive data sources discovered by ClueFinder, we were able to evaluate information leaks in 445,668 apps downloaded from 2 different app markets, gaining new insights into the way private user information (especially for those app-specific sensitive data) is accessed by third-party libraries. Across all these apps, our study shows that at least 118,296 (26.5%) disclose their customers' information to 3,502 libraries, which constitutes a privacy risk much more significant than reported by all prior studies. More specifically, we found that personal content has been extensively disseminated, including one's profile, installed app list, her social networking activities (e.g. profiles on Facebook and personal posts) and others. Particularly, among 13,500 most popular apps downloaded from Google-Play in 2015, 39.9% of them were found to expose user's information to 709 distinct third-party libraries, with each app on average sharing more than 7.6 private data items (e.g., address, profile, etc.) with at least 2 third-party libraries. Many of the libraries were found to indeed send collected user data out to the Internet, and only a few of them could be confirmed to only use such information on device (See Section V-B).

Also, such an *information exposure risk* (that is, using third-party libraries to process sensitive user data, which often leads to an unauthorized leak of the data to a third-party, as further showed in our adversary model) occurs when the app developer over-shares data for functionality enrichment or the third-party library aggressively gathers data through its hosting app. Among the top 100 libraries with the risk, 65% of them are non-ad libraries, such as Analytics, Social-Network utilities, etc., with hundreds of millions of installs through popular apps. A prominent example is *Tinder* (case study in Section V-C), a popular dating app that exposes its user's profiles and account name on *Instagram*, together with her instant locations to the library *Appboy* [6]. Also high-profile libraries like *ShareSDK* are given or actively acquire private information (e.g., user's social network profiles) unrelated to their missions (Section V-C). Not only do these findings confirm the long-standing suspicion that user information has been inappropriately disseminated through apps, but they also underline the scale and the breadth of such risks, which have never been fully understood before.

Contributions. The contribution of this paper are summarized as follows:

- *New technique for sensitive data source discovery*. We designed and implemented an innovative, semantics-driven technique for automatically recovering sensitive user data from app code, a critical step for leakage analysis. Our approach leverages semantic information of program elements, together with the unique program structures of their context, to accurately and also efficiently identify the presence of sensitive operations, which takes a step towards solving this long-standing challenge in app leakage analysis.
- *Large-scale exposure risk analysis and new findings*. Using our new technique, we investigated the potential information exposure to third-party libraries over 445,668 popular apps, a scale never achieved before in comparable studies. Our research brings to light the gravity of the problem, which has never been fully understood, and the channels through which such exposures happen, including over-sharing by app developers and aggressive data acquisition by third-party libraries. Further many high-profile apps and libraries were found to be involved in the information leaks. These findings help better understand this privacy risk and highlight the importance of data protection in today's software composition.

Roadmap. The rest of the paper is organized as follows: Section II presents the background of our research and assumptions we made; Section III elaborates the design of ClueFinder; Section IV presents the implementation and evaluation of ClueFinder and the supports it provides for a scalable leakage analysis; Section V describes our large-scale leakage study over 445,668 apps and our findings; Section VI discusses the limitations of our research and potential future research; Section VII surveys the related prior work and Section VIII concludes the paper.

II. Background

In this section, we lay out the background for our study, including privacy leakage analysis, the NLP preliminaries used in our research, and the assumptions we made.

App leakage analysis. Mobile users' privacy has long been known to be under the threats from the apps running on their devices. Information can be leaked both intentionally (often by malicious or gray app components) [42] or inadvertently (e.g., by leveraging the vulnerabilities in apps/mobile framewoks) [28]. Particularly when it comes to third-party libraries, what has been found is that many advertising (ad) libraries aggressively collect user data [30], [38] through different channels (un-protected APIs, privilege-escalation etc.), disclosing sensitive attributes like age, marriage status and work information to ad networks or advertisers. These findings, however, have been made on a small set of apps, due to the limitation to manually label and analyze privacy data sources and the data involved.

As mentioned earlier, automatic leakage analysis techniques have been widely studied, mainly through tracking "tainted" data flows across app code, from sources (e.g., the APIs for collecting GPS locations) to sinks (typically the APIs for network communication) [16], [23]. A well-known challenge for such analysis is identification of sensitive data sources, which mainly relies on the Android APIs with known sensitive returns, such as *getLastKnownLocation()* for locations, *getLine1Number()* for phone number, *AccountManager.getAccounts()* for account information, and others. Other

sources often need to be labeled manually. To facilitate data-source identification, tools like SUSI [35] can automatically recover from app code a large number of System APIs likely to import data. Less clear, however, is whether these APIs return sensitive information and therefore should be labeled as data sources. To capture such sensitive inputs, semantics of the imported content and the context of the related operations need to be studied. The idea has been used to find the sources on user interfaces, based upon the text content associated with sensitive user inputs such as "enter user name" and "password" [32], [25]. Even more challenging here is the labeling of the private data downloaded from the app's server or uploaded from its local repository. For example, when the user logs into her account, little context information is given during the importation of account data.

Natural language processing. ClueFinder leverages a set of NLP techniques to discover sensitive program elements and control false positives. Following we describe the key techniques used in our approach:

Stemming. Stemming is a process that reduces inflected (or sometimes derived) words to their stem, base or root forms: for example, converting "changes", "changing" all to the single common root "change". In our case, stemming helps us to find more semantic clues, particularly the program elements with prefixes and acronyms in their names: for example, the stem "addr" derived from "address" can match the variables names like *"user_addr"*.

Parts-Of-Speech(POS) tagging. POS tagging is a procedure to mark words as a particular part of a speech, based upon their meanings and context (relations with other words in a sentence, such as nouns and verbs). State-of-the-art POS tagging technique can already achieve over 90% accuracy [29]. Here we use POS tagging to determine whether a privacy-related keyword is actually a noun in the term or the sentence. For example, "address" in "address this problem" is a verb, just describing the action happening to another word, so it is not likely to represent a physical home address.

Dependency relation parsing. Dependency relation parsing analyzes a sentence, identifies the grammatic relations between different words and represents the structure of a sentence, based upon such pairwise relations, as a dependency tree. For example In the sentence *"Bell, based in Los Angeles, distributes electronics"*, the relation between "Bell" and "distributes" is described as *nominal subject*, where "Los" and "Angeles" represents a compound relation. In our research, such a dependency relation helps us to determine whether a specific privacy-related keyword is the dominator of its sentence, which is most likely to be the theme of the sentence.

Assumptions. The purpose of ClueFinder is to detect sensitive data sources from legitimate app code, covering those missed by all prior studies, particularly program elements related to the private user data imported from app servers. We do not consider deeply obfuscated programs that remove all semantic information from their program elements. Actually, Our study (see Section IV-B) shows that app developers tend not to obfuscate data-related code within their apps and the third-party libraries they integrate to avoid disrupting the apps' normal executions (e.g., causing a crash). As a result, we found that even moderately obfuscated code (e.g., through ProGuard) preserves a lot of semantic information: e.g., among all such apps discovered in our research, we found that over 50% of the method names are not obfuscated (Section IV-B) and over 98% of the apps still contain readable constant strings. Note that what we are interested in is unauthorized disclosure of sensitive data within an app to a third-party library, and therefore malicious apps covertly sending user data to the adversary are outside the scope of our study.

Further in our measurement study (Section V), we consider acquisition of sensitive user data by an untrusted third-party library to be an exposure risk. Even though this exposure does not necessarily mean that the data will be leaked to an unauthorized party, often the possibility of the leaks cannot be eliminated due to the complexity of data-flow analysis on these libraries.

III. ClueFinder Design

As mentioned earlier, although app code is semantics-rich, recovering truly sensitive data sources from the code is by no means trivial. Particularly, direct search for keywords does not work well, which misses many potentially sensitive tokens (e.g., prefixes, acronyms) in the identifiers of various program elements (variable, method, function, etc.). Also importantly, the presence of a single specific token does not necessarily indicates the operations on sensitive user data. In some cases, even when the token semantically looks perfectly relevant, for example the method *getPhoneNumberPrefix*, the corresponding program element may not touch sensitive data at all: in the example, the function does nothing but a format check on phone numbers. Thus, a precise semantic analysis is required to determine whether a token refers to a privacy-related activity. Further, the program elements carrying sensitive tokens may not carry private content (e.g., a constant string) themselves. They need to be linked to the true sources of private data through program analysis.

In this section, we show how the design of ClueFinder addresses these challenges and how the technique fares on real app code.

A. Design Overview

The idea behind ClueFinder is to quickly screen across a program to locate privacy-related tokens within program elements and then semantically evaluate the elements to drop false positives. Finally, a program structure analysis is performed to determine whether indeed each of these elements is involved in a sensitive data operation through a method invocation. Following we describe the architecture of this design and utilize an example to explain how it works.

Architecture. Figure 3 illustrates the individual components of ClueFinder and their relations. Our design includes a Semantics Locator, a Semantic Checker, a Structure Analyzer and a Leakage Tracker. Given an Android app, Semantics Locator first dissembles its code and identifies the program elements carrying sensitive tokens. These putative sensitive elements (string constants, variables, and method names) are then inspected by Checker, which uses NLP techniques to determine whether identified tokens are indeed the main subject of each element or its identifier's content (Section III-B).

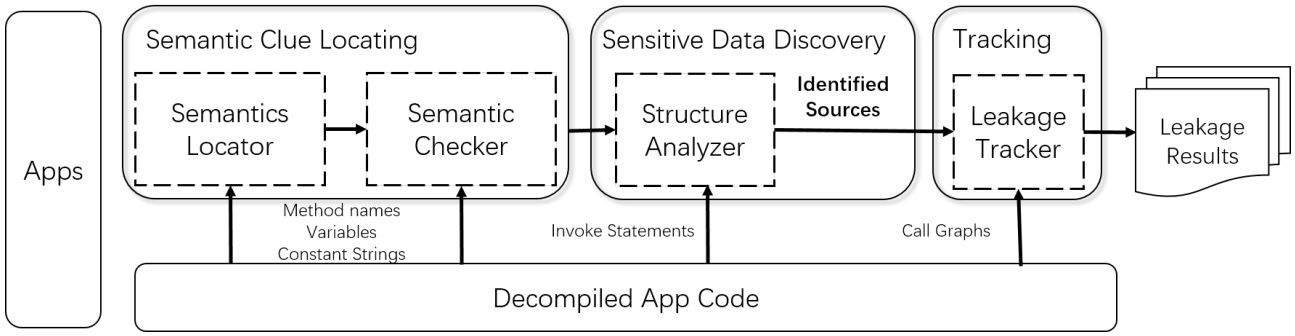

Fig. 3. Design of ClueFinder

Those meeting the standard further go through a program structural analysis, in which Analyzer classifies each function invocation statement involving an element as either sensitive or not. The reported sensitive ones are handled over to Tracker that incorporates the semantic information inside the app code into a data-flow and reachable analysis to trace the propagation of sensitive information.

Example. The example in Figure 4 shows the code snippet from *SnapTee*, through which one can design and purchase personalized T-shirts. Note that here we re-organize part of the app code for ease of illustration, while maintaining all semantics in the original (decompiled) app code. As we can see from line 25 to 29, the app first acquires a user's Facebook profile (line 26) and then sends a sharing post (line 28) to the user's Facebook account. After that, however, both the profile and the Facebook post are handed over to a third-party library function *trackShareEvent* (line 30).

To analyze the snippet, the Locater leverages a set of keywords representing 4 categories of sensitive data as identified by Google Privacy Policies [3] and prior research [39], [22], [32], together with their derived prefixes and acronyms through stemming to capture elements like *"home_addr"* (line 6), *getUserFbProfile* (line 26), *"I'm designing my own tees on my phone!"* (line 18), which all carry sensitive tokens *"home address"*, *"profile"* and *"phone"* respectively. The Checker then looks into these elements, picking out the ones like *getUserFbProfile*, given the observation that the token it involves (*"Profile"*) plays the role of a subject described by verb *"get"*. Also, for the long sentence in line 18, the Checker finds that *"phone"* actually not serves the theme (*"design tees"*) by the dependency relation parsing, and thus filters out this element as there is no indication it carrying private data. To further determine whether other elements left are indeed sensitive, their corresponding function invocation statements (e.g., line 5, 6, 8, 26) are inspected by Structure Analyzer, based upon their features: for example, statement in line 26 is a true positive since it returns a Json typed object, which could contain sensitive data, while the statement in line 5 is a false positive, since it only returns a boolean value to check whether the Json object contains a key with a name *"home_addr"*.

All identified statements are used as sources for a data-flow and reachable analysis performed by the Tracker. Such semantics-driven approach helps reduce the complexity of tracking propagation of sensitive data. For example, confirm a user profile leakage (a coarse-grained private data) from Line

```
1  ## In co.snaptee.android.utils.FacebookFunctions
2  Json getUserFbProfile(HashMap userBasicInfo) {
3    JsonObject userJson = UserBasicInfo.toJson();
4    ## Gather other user information
5    If(userJson .contains("home_addr")){
6      jsonObject.put("home_addr", this.homeAddr);
7    }
8    this.uri = jsonObject.get("userProfile_uri");
9    if(this.uri == null) {
10     throwNullPointerException("Profile URI is
           null", exception);
11   }
12   return jsonObject;
13 }
14
15 Builder shareToFacebook(String shareContent)
16 {
17   Builder builder = new Builder();
18   builder.setContentTitle("I'm designing my own
         tees on my phone!");
19   builder.setContentUrl(
         Uri.parse("https://snaptee.co/getapp"));
20   builder.setShareContent(shareContent);
21   Log.d("FacebookFunctions", "Try to invite FB");
22   return builder;
23 }
24
25 ## Getting user profile on Facebook
26 currentUser = getUserFbProfile();
27 ## Trigger sharing activity
28 shareToFacebook(shareContent);
29 ## Tracking user activity by invoking API from
       third-party library
30 trackShareEvent(currentUser,
       builder.shareContent);
```

Fig. 4. Overview code example

26, without analysing the actual code in this method (Line 2-13). The process ends when the related data is found to be accessed by a desired sink (e.g., a third-party library in line 30).

B. Semantic Clue Locating

Semantics Locator. To identify privacy-related data through the semantics of program elements, we first need to determine the set of data considered to be sensitive and keywords associated with them. Such information was gathered from multiple sources in our research. Particularly, we utilized 35 data items identified by Google Privacy Policies to be private content [3], together with additional 17 items reported by

prior privacy-related research [39], [22], [32]. For example, Financial Times (FT) [39] provides a calculator for evaluating the price of one's private data.

These items are organized by ClueFinder into 4 categories, including user identifiers, user attributes, location data and account information. In total, 121 keywords or keyword pairs are identified (see examples in Table I). Further, we use Word2Vec [31] to find more synonyms of these sensitive items. Also, the keyword set is extended using stemming (to find out their prefixes), with more similar texts extracted from 10,000 popular Google-Play apps (e.g., "addr"). This allows ClueFinder to capture as much sensitive semantics as possible from app code.

TABLE I. KNOWLEDGE BASE FOR PRIVACY-RELATED SEMANTICS

Category	Sample Keywords
User Attributes	first name, last name, gender, birth date, nick name, education, app list, device os, credit card, etc.
User Identifiers	user id, account number, access token, sina id, facebook id, twitter id, etc.
Location	latitude, longitude, lat, lng, user address, zip code, city, street, etc.
Account	account name, user name, phone number, mobile no, password, passwd, pwd etc.

For all elements in decompiled app code, ClueFinder uses word-splitting to break their names into tokens, using common delimiters (e.g., user_addr) and capitalized letters (e.g., *getUserFbProfile*). Then, it performs best-effort matching using its knowledge base (4 data categories and their representing tokens), searching for the tokens (keywords, prefixes and abbreviations) inside the identifiers of program elements. As a result, the elements involving privacy-related tokens are labeled for a more in-depth semantic analysis.

Semantics Locator finds out the elements with sensitive tokens in their names. This, however, does not necessarily mean that these elements are indeed privacy-related. As an example, in Table II, Index 1, the method *getStreetViewActivity* contains the sensitive keyword "street" but is actually unrelated to the user's location data. To remove such false positives, our approach runs Semantic Checker to further analyze the semantics of these elements.

Semantic Checker. Semantic Checker runs POS tagging and dependency relation parsing to get more in-depth semantic information from labeled elements. What we want to understand is whether a sensitive token actually serves as the "theme" of labeled content (element names or the content of a constant), which is more likely to indicate the presence of sensitive information, compared with the situation that the tokens are only used to describe other less nonsensitive terms (e.g., "street" in *getStreetViewActivity*). For this purpose, the Checker tries to determine whether the token is a noun and also characterized by the following dependency relations with its context terms in a phrase or a sentence:

- **Direct-object relation (Dobj)**: The direct object of a verb phrase is the noun phrase that is the (accusative) object of the verb: e.g., *getAddressFromServer* in Table II, Index 4. Here, the identified sensitive token with a noun POS tagger ("Address") has the *Dobj* relation with ("get"), indicating that this term is related to an access to location information. Other examples (1, 2, 3) are also presented in Table II.
- **Nominal subject (Nsubj)**: A nominal subject is a noun phrase that is the syntactic subject of a clause. This is a relation the Checker looks for in the absence of *Dobj* between an identified sensitive token and its context. For example, "business phone number selected", in which the sensitive token "phone number" is the topic of the sentence, indicating the presence of the information in its related program location. Example for such case is also presented in Table II, Index 5.
- **Negation modifier (Neg)**: The negation modifier is the relation between a negation word and the word it modifies. In our case, if the sensitive token found in the element appears with a *Neg* modifier, likely the element does not relate to sensitive content. E.g., "Do not input your password here".
- **Other relations**: When the sensitive token labeled actually has a dependent (*Dep*) or compound (*Compound*) or open-clausal complement (*Xcomp*) relation or other relations with its context (other words in the same element name or constant string), we found that the token becomes less of an indicator for the presence of private content, since the token in this case is no longer the theme of its context (the target of an access or the topic of a sentence). Examples for such relations are presented in Table II, Index 1, 2, 3, 6.

Using the relations above, Semantic Checker filters out the program elements involving sensitive tokens but less likely to be actually related to privacy content. These elements are then inspected by Structure Analyzer to further reduce false positives. In our implementation, the Checker was built upon Stanford Parser [29], a standard NLP tool for POS tagging and dependency relation parsing.

C. Sensitive Data Discovery and Tracking

Structure Analyzer. Even when a sensitive token plays a central role in the name of a variable or a method, or the content of a constant string, such program element may not necessarily relate to private content. For example, the statement in line 5 of Figure 4 talks about home address; however, the operation here is just checking whether the data object contains a key "home_addr". Another example is line 10 *"Profile URI is null"*, which actually is an output to explain an exception. So, to identify truly sensitive operations, not only do we need to check the semantics of the program elements' identifiers and constant content, but it is also important to look into the semantics of actual program operations specified by the statements involving these elements.

Serving this purpose is Structure Analyzer, which utilizes a set of program structural features to determine whether a sensitive-token related statement indeed touches private user content. In our research, we focus on method invocation statements, since sensitive user data are accessed by third-party libraries typically through method calls. To find such statements, the Analyzer first locates all method invocations (e.g., line 5, 6, 8, 10 in Figure 4) directly or indirectly related to a labeled program element (which involves sensitive tokens), and then extracts features from these statements to capture those accessed sensitive data. Specifically, when a method

TABLE II. EXAMPLES FOR SEMANTIC CHECKER

Index	Element	Description
1	getStreetViewActivity	As a negative example, "street" only holds a *Compound* relation with "Activity", the *Dobj* relation here is between "get" and "activity".
2	getLocationUpdate-TimeIntervalInMillis	As a negative example, "Location" only holds a *Compound* relation with "Interval", the *Dobj* relation here is between "get" and "Interval"
3	"I'm designing my own tees on my phone"	As a negative example, "Phone" only holds a *Nmod:poss* relation with "my", and a *Nmod:on* relation with "design". The *Dobj* relation here is between "design" and "tee".
4	getAddressFromServer	As a positive example, "address" here is with POS tagging "NN", and holds a *Dobj* relation with verb "get".
5	"Username must be in valid format"	As a positive example, although there's not *Nsubj* relation in the sentence, "Username" holds a *Dobj* relation with "Format".
6	new_friend_num	As a negative example, "Friend" only holds a *Compound* relation with "num".

name is labeled, all statements that trigger the method are considered to be potential sources of private information. For labeled variables and constant strings, the Analyzer performs a data-flow analysis on them to identify all the invocations that take the elements or their derivatives as parameters. All such statements are then inspected for their program structural features.

Our key observation is that when labeled elements are involved in data read or write operations, almost always the operations are related to sensitive information, with the source of the information being the element when it is a variable, another variable in the same method call when the element is a constant, or the return value of the call when it is a method. Leveraging this observation, our approach analyzes how these elements are used in an invocation statement to seek evidence that such sensitive data operations indeed take place. Such evidence could be as simple as the presence of keywords such as "get", "put" in a method name (e.g., *getUserFbProfile* in Figure 4). It can also be the return of a data-typed object from a method call: e.g., *getUserFbProfile* returns a Json object (line 12 in Figure 4). Another example is the pattern of using different types of data together: e.g., a constant string (key) often appears in front of a string variable (value) in a method invocation; an example is line 6 of Figure 4. These features are summarized as follows:

- **Method name**. As mentioned earlier, a feature used in our research is whether a method name in a labeled statement contains a specific token representing data operations, such as get/set/put/add/insert/delete/remove/read/write/save.
- **Parameter type**. We also look at the primitive data-types of the parameters in a method invocation, which indicates the presence of data operations. Examples include String, HashMap, Json, etc.
- **Return type**. The return type of a method call also provides evidence for the presence of data operations: e.g., a data read brings back a result in String, HashMap, Json, etc.
- **Base value type**. Many data operations happen through specific Java class libraries. Therefore, for those statements which contain a base value (e.g., *hashMap* in method *hashMap.put(key, value)*), the class type of the base value can also help differentiate data access from other operations. For example, in Figure 4 line 6, the *Json* class for *jsonObject* is used to process data while in line 21, the *android.util.Log* class for base value *Log* does not relate to data use.
- **Constant-variable pattern**. Also useful to identification of sensitive data operations are the patterns of constant-variable parameter combinations in method calls. For example, the first parameter of *hashMap.put("user", $u)* is a constant and the second is a variable, which is a standard key-value combination for a data-processing method call. In another example *hashMap.put("user", "default")*, its parameters are all *String-Constant*, and thus the call does not indicate the existence of data access.

On top of such features, Structure Analyzer runs a Support-Vector Machine (SVM) classifier to determine whether a given statement indeed involves private data. The classifier was trained using 4,326 statements randomly selected and manually labeled from 100 apps, as elaborated in Section IV.

Leakage Tracker. The statements together with privacy-related semantics recovered by ClueFinder are treated as the actual "sensitive" sources for detecting information leaks. Specifically, Leakage Tracker extracts data-typed objects within the statements from their parameters or return values, and then performs a data-flow based taint analysis on these objects. The purpose of this analysis is to find out whether sensitive data flows get into the sinks that indicate leaks of the information to unauthorized third parties. Ideally, one may expect that such a sink is an API used by an untrusted library to send tainted data out to the Internet, as did in prior research [37]. In practice, however, tracking tainted flows across library code is often too heavyweight and less precise, particularly for a static analysis important for evaluating a large number of apps. Therefore, in our research, we instead looked for the presence of an *exposure risk*, when the tainted data flow into an untrusted library, since in this case, the data is no longer safe and its content could be disclosed to the third-parties through various channels hard to capture by the existing technologies (i.e., cover channels).

What is unique for ClueFinder is its utilization of semantics to enhance the taint analysis, which enables more efficient detection of the exposure risk. For example in Figure 2, even without analyzing the code from Line 2 to 5, the semantics of the method invocation at Line 14 (e.g., the constant "last_location" involved) immediately reveals the involvement of sensitive content in the function's return value (*basicInfo*).

In this way, we can quickly determine whether private data are under the exposure risk, avoiding more expensive data-flow analysis.

IV. Evaluation of ClueFinder

In this section, we first describe our experimental settings for evaluating ClueFinder, and then report its effectiveness and performance. Also, we compared our approach with prior work, which demonstrates that ClueFinder outperforms the prior approaches in terms of sensitive data discovery.

A. Experiment Setting

We implemented ClueFinder in Java (1,604 LOCs) and Python (609 LOCs). Our implementation extends the FlowDroid framework for analyzing decompiled packages in the Jimple format (an intermediate expression for analyzing DEX code). Note that since FlowDroid renames all local variables (like "$r1", "$r2") when decompiling app code, current implementation of ClueFinder only utilizes global variables like static fields. ClueFinder also utilizes the Java implementation of Stanford Parser [29] for its NLP analysis. Its Structure Analyzer component extracts the features from the Jimple statements and runs the Python implementation of SVM from Scikit-Learn [11] to train the classifier. All our experiments were conducted on a 32-core server, with a Linux 2.6.32 kernel and 64GB memory.

Training data. The classifier was trained using a *labeled set* of 4,326 statements (half positive and half negative) which were manually labelled by two Android experts from 100 popular apps. Specifically, in this manual-labelling process, we first randomly selected 100 apps from Google-Play (crawled in August, 2016) based on the top-popular list during that period. Then, we automatically extracted all statements involving privacy tokens from these apps by Semantics Locater and Checker (See Section III-B), and let each of the expert to identify if the given statements contain private data or not. To create a more precise training set, each statement was labelled as either positive or negative only when both of the two experts give the same result. In total, we collected 7,354 labelled statements, including 5,191 positive samples and 2,163 negative ones. Since SVM classifier usually gets better results under balanced training set [21], we used all negative statements, together with the same amount of positive statements by random selection from labelled data as our training set. As a result, the total amount of our training set is 4,326.

B. Effectiveness

In our experiment, we first ran a ten-fold cross validation on our *labeled set* (with 4,326 labeled statements in 100 apps). ClueFinder achieved a precision of 92.7%, a recall of 97.2% and a F1-Score of 94.8%. Since our training set is randomly picked, the effectiveness of the classifier should carry over the entire app code with high probability. Also, we employed another manual validating process, by running ClueFinder over another 100 randomly selected apps crawled at the same time as an *unknown set*. Our manual validation showed that 320 out of 3,775 statements are false positives, which gives a precision of 91.5%. We did not get the recall in this manual validation due to the lack of ground truth (it is rarely possible to manually go through all code in the dataset to identify which of them are indeed sensitive sources).

The total analysis time for the 100 *unknown set* were 97 minutes (less than 1 minute per app). Such a performance level enables ClueFinder to process a large number of apps, as we did in our research (Section V).

False positives and false negatives. Most false positives reported were caused by rare cases that were not covered by our labeled set: as an example, in Figure 5, the constant parameter for the method *saveEvent* includes the sensitive term "access token", which however turns out to have nothing to do with the variable $r1$, an object with an "Event" type. Also, in some cases, even the program structure does not offer sufficient information for determining whether a statement involves sensitive information. For example, in line 2 of Figure 5, the method *saveAppKeyAndAppSecret* contains sensitive tokens like "Key" and "Secret"; however, such private data only appears within the method and the invocation statement is actually nonsensitive.

When it comes to false negatives, again many problems were introduced by the outliers. For example, the statement at line 4 of Figure 5 returns an integer value to encode the gender information (1 for male and -1 for female), which does not meet the expectation that the gender data is supposed to have a string type. Another source of the problem is the incomplete knowledge base: some sensitive terms, such as "lon", "father's name" and "mother's name", are not considered keywords for sensitive content; as a result, what ClueFinder discovers is only a subset of truly sensitive data items.

```
1 void saveEvent("init", "put access token to
      extras", $r1);
2 Umeng.UMTencentSsoHandler: void
      saveAppKeyAndAppSecret();
3 Java.Util.HashMap<Object,Object>.put("username",
      $r1);
4 Integer gender = getUserGender(user);
```

Fig. 5. False positive and false negative samples

Code obfuscation. As mentioned earlier, ClueFinder is not designed to analyze deeply obfuscated code with all its semantic information removed. This, however, does not mean that our approach cannot tolerate any obfuscation and can be easily defeated by the tools like ProGuard [9]. Actually we found that a significant amount of semantics is preserved in moderately obfuscated code, e.g., that protected by ProGuard, and therefore can still be analyzed using ClueFinder. For example, Figure 6 shows a code snippet obfuscated by ProGuard. As we can see here, strings (e.g., "ReportLocation" in line 1), parameter types (e.g.,"String" and "Object" appeared together in line 2) and API calls (e.g., *JSONObject.put()* in line 4) all carry meaningful content, which can be leveraged as features by ClueFinder's classifier to determine the presence of sensitive data sources[1]. In our research from the aforementioned *unknown set* with 100 randomly selected apps, we found

[1]Note that even though BidText could also utilize constant strings, it does not work on other code features and therefore will be less effective in analyzing such code, as we further elaborated in the comparison with ClueFinder

that 11.3% (426/3,775) of the statements in these apps were obfuscated. Nevertheless, sensitive data sources and exposure risks within these apps were all identified by our approach, since our classifier leverages a whole set of features that cannot be easily obfuscated, such as system-level parameter objects and return values like String, Json, etc.

```
1 $r0.com.*.sdk.ei: void
      a(String,Object)>("ReportLocation", $r3)
2 $r1 = staticinvoke <com.*.SharedPref: String
      b(Context,String,String)>($r0, "Gender", $r1)
3 $r4 = virtualinvoke $r3.<com.*.bean.ay: String
      c(String)>("user_password")
4 $r7.<JSONObject: JSONObject
      put(String,Object)>("cust_gender", $r2)
```

Fig. 6. Samples of partially obfuscated statements (in Jimple format) identified by ClueFinder

Such semantic information is preserved due to a few practical constraints in code obfuscation. Specifically, system-level methods, as discovered by SUSI, cannot be easily obfuscated and their meaningful names and parameters therefore are retained by the tools like ProGuard. As an example, 98% of constant strings in our unknown set are human-readable. Also, we found that app developers tend to avoid obfuscating data-related modules (e.g., those containing GSON objects) and third-party SDKs, since improper changes to these program elements could easily introduce errors to program execution or even cause a crash. As an example, GSON utilizes reflection at runtime to dynamically map JSON objects to classes, constructing properties based upon matching strings discovered from the objects with keywords; this approach no longer works when the string such as "model.name" is replaced with "a.b" by ProGuard. Further, third-party frameworks (e.g., Inmobi [4]) and SDK interfaces are rarely obfuscated, to make sure that the developers can easily incorporate them into her app code.

C. Comparing with Prior Approaches

API-based labeling. As mentioned earlier (Section II), prior work SUSI [35] can automatically discover hundreds of sources from various Android System APIs. However, it often cannot determine whether a source is indeed sensitive. For example, *json.get("password")* becomes related to sensitive content only because its parameter reveals that the method returns password.

ClueFinder is designed to identify such sensitive sources from their context. In our research, we randomly selected 15 popular APIs and extracted 10,116 statements involving them from 100 randomly chosen apps. Among all these statements, ClueFinder detected 2,266 sensitive data sources. As a result, over 77.6% (7,850) of statements which SUSI found turning out to be false positives (not sensitive). Given the effectiveness of ClueFinder that already discussed before (with 92.7% precision and 97.2% recall), the comparison result indicates that our approach is much more effective in finding truly sensitive data sources compared with SUSI.

UI-based labeling. Prior approaches like UIPicker [32] and SUPOR [25] can identify sensitive data from an app's UI elements (e.g. an input field). However, these elements are only a subset of private data sources in an app. In contrast, ClueFinder is capable of finding all sensitive sources, including not only UI elements but also imports of private data from servers. In our study, from the aforementioned *unknown set* with 100 randomly selected apps, we manually identified 892 unique UI elements related to private inputs (e.g., username and password in UI). These elements are all the prior approaches could find. Then, we ran ClueFinder over the 100 apps which reported 2,388 unique sensitive data sources. Our further manual validation over these sources showed that in most cases, ClueFinder identifies all the UI sources. What's more, it identifies 2 times more non-UI sources missed by approaches like UIPicker and SUPOR in total.

Semantics-based taint tracking. Similar to ClueFinder, BidText also searches constant strings inside programs for sensitive keywords. However, BidText is more focused on its unique bi-directional taint analysis than semantics-based sensitive source discovery. It does not work on variable, method names, prefixes and abbreviations of keywords, nor does it evaluate grammatical dependencies among semantics tokens except the negative relation. In our research, by setting up these two approaches with basically the same settings for data-flow analysis (we implemented an intra-process, flow-sensitive analysis with sinks to HTTP network), we compared our implementation of ClueFinder with the released version of BidText [2], in terms of precision, coverage and performance in discovering sensitive sources.

As Table III shows, among the 100 popular apps in the *unknown* set, ClueFinder reported 50 (44.6%) more sensitive sources than BidText (162 true positives vs. 112), resulting in a much higher coverage than BidText. This is mainly due to ClueFinder's in-depth NLP analysis for understanding code semantics, as well as the utilization of code structure for locating private data. For example, ClueFinder found 32 sensitive sources using semantic information from method names, which BidText could not handle. Also, ClueFinder has a more detailed knowledge base (characterized by not only keywords but also meaningful prefixes and abbrievations), and its semantic locating mechanism (in Section III-B) enables it to capture more privacy-related semantics: e.g. "addr" for "address". BidText utilizes only a fixed keyword set, and matches these keywords from app code by human-defined regex expressions.

Also, ClueFinder reduces the false positive rate compared with BidText (8.5% vs. 14.5%), because our approach utilizes more grammatical relations and program structures to control false positives, while BidText only drops the labled strings involving negative description. Other strings, such as 1, 2, 3 in Table II, will be falsely reported. Further, since BidText only evaluates constant strings and most of them do not contain complicated expressions, our approach only lowers down the false positive rate by 6% compared with the prior approach. It is important to note that the major strength of ClueFinder is it expands the types of sensitive sources that can be discovered. In this perspective, the advantage of our approach is significant, detecting 44.6% more true positives.

Finally, ClueFinder outperforms BidText in performance (1.86 times faster), due to its lightweight semantics-based

leakage analysis, which largely avoids the expensive data-flow analysis. In the meantime, we acknowledge that BidText's unique bi-directional dataflow technique could help it more accurately track some information leaks ClueFinder misses, given that the focus of our approach is just sensitive source discovery.

TABLE III. COMPARISON WITH BIDTEXT

	BidText	ClueFinder
Detected sensitive data	131	177
Num. of false positives	19	15
Avg. Analysis Time (Sec)	97	55
Precision	83.5%	91.5%

V. LARGE-SCALE LEAKAGE STUDY

In this section, we report our measurement study over 445,668 real-world apps, which analyzed their privacy leakage to third-party libraries. Note that although ClueFinder is capable of detecting all kinds of private data within an given app code, here we just focus on the findings related to the sources missed by the prior research, since more conventional sources, such as API-based imports of IMEI, IMSI and GPS locations, have already been studied before [22], [37], [40]. Our research brings to light the pervasiveness of the exposure risk (disclosing sensitive user data to third-party libraries) and interesting cases never reported before.

A. Measurement Settings

Exposure risk. As mentioned earlier (Section I), in our measurement study, we looked for the exposure risk, that is, leaks of sensitive user data to third-party libraries. We focus on this risk instead of the library's export of sensitive data to the Internet because the latter is more difficult to detect through a static analysis (necessary for evaluating a large number of apps), in terms of performance and accuracy. Also, once an untrusted library obtains private data, it often can manage to send the data out through cover channels without getting caught. Therefore, in our study, we just conservatively considered that information leaks could happen whenever the untrusted library gets access to the sensitive data.

App gathering. As Table IV shows, our datasets are crawled from 2 different Android markets: the official Google-Play market and a third-party market (Tencent App Store). Each app in these datasets has a unique MD5-Hash to make sure there's no overlapping between different datasets. Among them, apps in the *Play-15* dataset were selected according to the top app list provided by the Google-Play website, and those in the other 3 datasets were randomly crawled from their markets. In this way, we can better understand how data leaks to third-party libraries happen in both popular and ordinary apps.

Implementation for Leakage Tracker.

Specifically, serving the purpose of detecting privacy leakage to third-party libraries, Leakage Tracker in ClueFinder (Section III-C) went through all the invocation statements reported by its previous module Semantic Checker, and conducted a inter-procedure data-flow analysis over the identified data objects. Meanwhile, it picked out those statements either inside a third-party library or calling the library's methods. As an example in Figure 2, if the method contain HashMap object "basicInfo" flows to an API of a third-party library, immediately we conclude that the user's location data are exposed to the library by this statement.

To this end, we checked whether the package or class name of the identified statement is different from that of the app, using its first two prefixes, e.g. com.facebook for com.facebook.message, which indicates that the statement is either inside a third-party library's code or involves the library's method. Although this treatment is a bit coarse (e.g., which cannot distinguish the ad library *com.facebook.ads* from the analytic one *com.facebook.analytic*), it is still informative for us to determine whether private data have been accessed by a third-party library or by the app itself. Further we verified that such a statement is not dead code through a standard reachability analysis: that is, building call-graphs from the app's entry points to confirm that indeed the target method invocation can be reached. Note that this treatment can miss some information leaks, however, it is sufficiently accurate for detecting most leaks to third-party libraries because most of such invocations could be the interfaces between a library and its hosting app, and also lightweight, which is important for a large-scale study.

We utilized the experimental setting described in Section IV for the measurement study. During the experiments, each dataset was processed by 8 concurrently-running processes, with a 20-minute timeout set for each app. Overall, our 32-core server took 710 hours to go through all 445,668 apps, with 45.88 seconds each on average. Among all these apps, 32,533 (7.3%) could not be successfully analyzed within the timeout window.

B. Measurement Results

Landscape. As can be seen from Table IV, among all 445,668 apps, ClueFinder totally discovered 118,296 (26.5%) leaking private user data to 3,502 third-party libraries[2]. On average, each app exposes 8.07 data items (e.g., an identifiers, full name, location, etc.) to 1.97 libraries. This indicates that such information exposure is indeed pervasive (over 26.5% of all the apps analyzed). For example, when the user logs into an app with her Facebook account, her Facebook profile could be sent to an ad library for marketing, and to an analytical library to track her online activities. Also, for all discovered 3,502 libraries accessing user's private data, averagely each of them collects 2.45 data items, including not only different identifiers such as Facebook id, but also other information like her various attributes, for the purpose like targeted advertising.

Particularly, the *Play-15* dataset, with selected 13,500 most popular Google-Play apps, was found to have 39.9% of its apps leaking out user data. As illustrated in Table V, such data are

[2]To avoid including outliers (e.g., an obfuscated package name) as a third-party library, we first exclude those extremely short package names (e.g., com.a.ab) which obviously to be obfuscated. Meanwhile, we define a threshold=10 to decide whether a package name surely presents a third-party library. The threshold is the number of total appearances of a package name in our whole dataset. Also, we exclude common social network libraries (e.g., Facebook, Twitter, Weibo, etc.) since most of private data in such libraries are originated from themselves.

TABLE IV. OVERALL LEAKAGE STATISTICS

DataSet			Affected Apps				Affected Libs	
	Collect Time	Total Apps	# Apps	% Apps	Avg.Items/App	Avg.Libs/App	# Libs	Avg.Items/Lib
Play-2015	Nov.15 - Dec.15	13,500	5,385	39.9%	7.6	2.83	709	2.45
Play-2016	Jul.16 - Aug.16	71,686	16,310	22.8%	5.26	1.32	1,011	2.36
Tencent-2015	Feb.15 - Apr.15	169,051	44,392	26.3%	7.55	1.64	2,315	2.43
Tencent-2016	Jun.16 - Jul.16	191,431	52,209	27.3%	9.53	2.1	3,097	2.33
Total	Nov.15 - Aug.16	445,668	118,296	26.5%	8.07	1.97	3,502	2.39

uniformly distributed across several categories (user attributes, user identifiers, account information and location data), with each app exposing 7.6 data items to 2.83 third-party libraries on average. Compared with randomly selected apps in *Play-16*, these top apps apparently expose more information. This indicates that popular apps extensively disclose all kinds of private user information to multiple libraries within a single app. Further, by manually looking into the code of 100 randomly selected apps identified by ClueFinder, we found over half of the flagged method invocations (53.1%) are related to HTTP connections (e.g., an HTTP post where its parameters contain privacy-related contents). Also, our runtime verification by intercepting the network traffic of these apps confirmed that 59 out of 100 apps are indeed leaked private data to the servers of different third-party libraries. Note that the actual leakage scale should be higher than what we observed. We didn't see the traffic for the other 41 apps since most of them require further manual steps, e.g., logging in or even pre-registering an account. Additionally, since some libraries encode or encrypt their traffic, the leakage cannot be directly confirmed even when the app was well-explored.

TABLE V. LEAKAGE RESULTS BY PRIVACY CATEGORY IN PLAY-15 DATASET

Category	Apps (%)	Avg.Items	Libs	Avg.Libs/App
User Attributes	4,928 (36.5%)	4.19	401	2.38
Account	2,444 (18.1%)	2.47	210	1.81
User Identifiers	5,157 (38.2%)	3.43	659	1.69
Location Data	4,307 (31.9%)	2.77	379	1.84
Total	5,385 (39.9%)	7.60	709	2.83

Further, by comparing *Play-16* with *Tencent-16* in Table IV, we observed that individual apps on the un-official market (Tencent) tend to integrate more third-party libraries (1.32 vs. 2.1). Since the security vetting process in app-market like Tencent usually not be as strict as Google, apps in such un-official stores tend to be more aggressive in collecting private data. Another observation is that although the amount of third-party libraries has a reasonable increment, by comparing from same dataset crawled in different periods (*Tencent-15* and *Tencent-16*, column 7-9 in Table IV), the overall apps in the market tend to have almost identical leakage scale (see column 9). This indicates such privacy leaks to third-party libraries is a long-standing problem without noticed, due to the lack effective discover tools like ClueFinder.

Library distribution and leakage patterns. ClueFinder discovered that 3,502 libraries access private user data. To understand what these libraries are and how they collect sensitive information, we took a close look at the top 100 most popular libraries from our datasets. Table VI summarizes our findings[3], where column 2 shows the percentage of libraries in different category, and column 3 shows the percentage of apps contain one of such libraries from all apps involved in privacy leakage. As we see here, most of those gathering user data turn out to be ad and analytical libraries (e.g., Inmobi, AppBrain, etc.). These libraries do not enrich their hosting apps' functionalities but constitute the major source of information leaks.

TABLE VI. DISTRIBUTION OF TOP 100 THIRD-PARTY LIBRARIES FROM PLAY-15 DATASET.

Library Category	% Libs	% Apps
Ads	35%	80.7%
Analytics	27%	68.9%
App Dev Framework	26%	36.9%
Utils	21%	16.4%
Social Network	14%	6.2%
Game Framework	11%	9.6%

From the ways these libraries interact with their hosting apps, we can see that they are either *given* private information by the apps through API calls or actively *harvest* information (such as transferred location data, installed app list on device and timestamps for specific events) from the apps, without the app developer's awareness. ClueFinder differentiates these two scenarios by looking at where the identified sensitive statements are located: if the statement is inside the hosting app's code, clearly the app's developer intends to pass information to a library, often for enriching the app's functionalities or communicating with advertisers; Otherwise, when the statement is found in the library code, apparently the library collects user data without proper authorization. For example, the library starts a service in background when app invokes one of its public interface. We show the breakdown of these patterns in Figure 7. Also, we present our findings about these cases in Case Study (Section V-C).

Leaked content. Table VII presents prominent examples for the data items exposed to the third-party, as discovered by ClueFinder. It does not come as a surprise that several kinds of identifiers are disclosed (e.g., facebook id), often together, since they are often used in combination to track a user, even when she re-installs the app or changes her device. Regarding user profiles like gender and nick name, most of them are from social networks like Facebook. Due to the extensive use of mobile single-sign-on, once the user authenticates an app with her social network account, an authorization of profile access on social network has also been granted to the app.

[3]One library can have multiple functionalities in different categories, as we can see from the table.

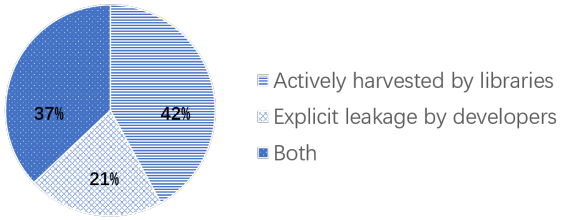

Fig. 7. Distribution of leakage patterns in detected libraries

As a result, some of such data are open to other third-party libraries the app integrates. We present a case study about such a leakage in Section V-C.

Exposure of user locations is another major source of leaks captured by ClueFinder. Unlike prior findings that location data are read directly through System APIs (e.g.,*getLastKnownLocation* [42]), interestingly most such leaks reported by ClueFinder are caused by retrieving location-related data from other, less sensitive sources, such as an app's persistent storage (e.g., *SharedPreference*, local database, etc.). This new location acquisition strategy could attribute to the enhancement of privacy protection on today's Android devices. Increasingly, iOS-style runtime access control has been adopted and even more fine-grained control [10], such as asking for the user consent for *every* location access. As a result, third-party libraries tend to avoid frequently invoking the sensitive System APIs, even when the app indeeds has the location permission, and instead reuse the location data collected when the app has a legitimate reason to do so, e.g., when the app is just launched to the foreground. Also we observed that some libraries even try to gather other location-related information that does not need a location-related permission to access. Examples include the BSSIDs of Wi-Fi hotspots, which can be used to infer locations, as reported by the prior research [41].

It is also worth noting that 12.0% of flagged third-party libraries gather the information about installed apps on a device. The list of installed apps can be used for different purposes. For example, Ironsec [5] claims on its website, "Using this platform, we're able to accurately predict what app a person will want to install next". As another example, on the website of MoPub [8], it states "User targeting allows you to target users that have or don't have specific applications". Besides, We found that some libraries, such as *co.inset.sdk* [1] and *ShareSDK* [12], even persistently monitor app installation, collecting package names and other data such as locations, app-usage time etc. As reported by the prior studies [37], [18], [22], linking installed apps to public auxiliary information can lead to violation of user privacy: for example, the presence of a gay dating app exposes the user's sex orientation.

C. Case Study

Here we present some high-impact cases discovered in our research with runtime verification by intercepting their network traffic. As showed in Table VIII, these cases involve high-profile apps as well as popular libraries.

Case 1: Deliberate harvesting. *The-Paper* is a popular app focusing on Chinese political news with 16 million downloads [14]. Like many other news apps, it allows its users to share news they read with any social networks (e.g. Weibo) or friends using these networks. Inside the app, this function is actually provided by *ShareSDK*, which acts as a syndicator that integrates multiple social networks. The library is supposed to serve as a "proxy", accepting the user's sharing request and forwarding the content to the intended social network platform (e.g., Weibo). However, ClueFinder found that this library also accesses detailed user profile data, which obviously goes beyond the library's stated functionality. Our manual review of the app code further shows that *ShareSDK* actually deliberately collects much more private data than necessary. By utilizing the authorized permission for Sharing, *ShareSDK* also gains the ability to read other data about the user on its social network. As a result, it collects all user profile information like true name, gender, verify status, even education background information. Also, it records such data and send them to its own server. We list part of the sensitive information *ShareSDK* collects in Table IX.

TABLE VII. SELECTED PROMINENT LEAKAGE SAMPLES

Item	% in Detected Apps	% in Detected Libs
location	86.6%	90.5%
facebook id	26.7%	32.5%
gender	22.5%	32.6%
app list	15.9%	12.0%
nick name	13.1%	10.9%
oauth token	10.4%	12.3%
date-of-birth	4.2%	3.1%

TABLE VIII. APP & LIBRARY USED IN CASE STUDY

App Name	Num.of Installs	Library	Num.of.Apps
The-Paper	16 million	ShareSDK	13,468
Tinder	50-100 million	AppBoy	419
SnapTee	10-50 million	MixPanel	7,284

TABLE IX. PRIVATE DATA WHICH COULD BE COLLECTED BY SHARESDK

App Info	top-task app list, app start timestamp, app end timestamp, new install app, new uninstall app info, etc.
Social Network Info	
Weibo	weibo id, nickname, true name, verified reason, gender, sns url, resume, friendlist, shared posts, latitude, longitude, liked posts, etc.
Facebook	facebook id, nickname, gender, birthday, sns-url, friend list (including accessible friend info), verify status, education (school name, type, year), work (company, employer, start & end date), etc.
Others	tumblr, dropbox, pinterest, line, tencent qq, tencent qzone, wechat (friend list), twitter, net-ease microblog, evernote, google+, etc.

Our further investigation shows that *ShareSDK* is widely integrated by most popular Chinese apps, each with more than millions or even billions of downloads. However, the library's privacy harvesting behaviors have never been reported

and therefore are totally oblivious to the app users. Using such profile data, this SDK can track a user and identify her personal characters from different vectors (e.g., what she "liked" on Weibo, what posts she marked as favourite), as well as her own social connections (e,g., friend list, followers, working company, etc.). Also, since the library records the user's operations on its hosting apps (what she shared to social network) and therefore knows a lot about her, for example, her political stands.

Case 2: App data over-sharing. Tinder is a famous dating app, with around 50-100 millions of downloads on the Google-Play store [15]. The app integrates AppBoy [6] to collect statistics information about its users. Each time a user takes a certain action within the app, Tinder synchronizes its action record to AppBoy, together with many sensitive data about the user. As illustrated below, when a user refreshes a window to display other nearby users, Tinder sends her precise location, bio information, dating targets, as well as her name on Instagram to AppBoy. All such information disclosures are unknown to the user, as showed below.

```
{"package_name":"com.tinder",
"extras":{"device":{"push_token":"..."},
"user":{"Seeking Distance":50, "gender":"f",
"Account Creation
    Date":"2017-05-1*T16:56:32.163Z",
"Seeking Gender":1, "Has Work Info":true,
    "Has Education Info":true,
"Instagram":"Susan_***", "Has Bio":true,
"Number of Profile Photos":15},
"sessions":[{"guid":"...",
    "start_time":1.479401816693E9,
"events":[{"d":{"ll_accuracy":19.80900,
"longitude":-8*.4778, "latitude":3*.1615},}]}
```

Case 3: Social network data over-sharing. SnapTee (co.snaptee.android) is a popular T-Shirt design app that allows users to buy tees either customized by themselves or by other designers. Also, users can share their designs with various social networks (e.g., Facebook, Twitter) through the app. We observed that when a user connects her SnapTee account with a social network, Snaptee updates her profile including full name, email, account ID and other information collected from the social network. Further, the app passes all such profile data to a data analytic library MixPanel [7]. From MixPanel's website, we found that the library is designed to "understand who your users are, see what they do before or after they sign up". However, the user is kept in the dark when such data collection and sharing happen. Following is the information Snaptee shares with MixPanel.

```
{"$set":
  {"$username":"p***t",
      ..,"$email":"li**v@gmail.com",
   ..,"$first_name":"John",
       "$last_name":"Smith",
   "Twitter":"795**16"},
       "$token":"f81d***cdf96",
   "$time":"1479324910201",...}
}
```

VI. Discussion

Android users have been suffering from privacy leakage issue for a long time. Fortunately, with the advent of ClueFinder, the issue will be mitigated because developers can track various types of sensitive data in a more efficient way. Specifically, the combination of semantic-based and code structure based analysis makes precise localization against private data possible, whereas traditional methods using fixed APIs can not label. With the help of more sensitive sources found by ClueFinder, existing privacy leakage analysis tools can be improved by taking advantage of better precision and wider coverage of users' sensitive data. For example, ClueFinder can be employed with both static and dynamic taint analysis [23] [16], by assigning data objects within the statements as sensitive sources. It can also be applied in various access control mechanisms [20] [19] for fine grained control over the sensitive data within the app.

Our measurement study against 445,668 apps helped us to get a better understanding to the privacy-leakage issue of Android apps. Although most of legitimate apps provides information about how they manage users' private data, including what from third-party libraries, their vague descriptions are proven to be weak and unpractical for effective protection to user's private data [43]. Our findings including the over-sharing and third-party aggressive data collection highlight the necessity of fine-grained access controls over these private data. For example, alerting users with detailed private data leakage information by third-party library at runtime.

Admittedly, ClueFinder does have limitations. For instance, since ClueFinder heavily relies on semantics in app code to discover possible private data, obfuscation may help adversaries to evade our analysis, as semantics including strings or method names are helpless for those cases. However, as mentioned in our evaluation (See section IV), given that most apps do not obfuscate the entire code base, ClueFinder is still a very practical approach of discovering private data at a large-scale.

Besides, the effectiveness of ClueFinder can be further improved from several aspects. (1) Find more semantic resources in the app to improve the coverage of ClueFinder. Current implementation of ClueFinder only consider semantics from method names, variable names and string constants. Other information like package name (e.g., *facebook.userInfo.facebookUserProfile*) may also provide abundant semantics. (2) Find more features in app code to improve the precision of the SVM classifier. E.g., features at the caller/callee of the candidate statements may also help to decide if it contains sensitive data.

What's more, our measurement results for privacy leakage (Section V-B) indeed tell if a specific private data have been accessed by third-party libraries, while the results need further pruning: First, the measurement did not confirm if all such private data accessed by third-party libraries are indeed leaked out at a large scale. Instead, as mentioned in Section V-B, we manually validated a small set of apps and confirmed over half of them involved in privacy leakage, as a lower-bound of the actual leakage scale. Although it is possible to give a further static taint analysis by assigning network APIs as the final sinks, the result may not be feasible due to the fundamental limitation of static analysis approach (e.g.,

heavy-weight and less precise). Further more, our system ClueFinder, was designed to find more sensitive data sources. Serving this purpose, our approach achieves a precision of 91.5% (Section IV-B). The measurement of privacy leakage to third-party libraries is just a demonstration of how our technique can be used. Second, our current approach can not automatically distinguish if a given access by third-party libraries is reasonable, though our manual analysis shows that most of such access to private data is suspicious. Further analysis could utilize semantics from app UI, app descriptions and many other possible sources to determine if such access is benign or malicious. An access is regarded as malicious only if there is no matched UI or app description for the private data access in apps.

VII. Related Work

Privacy leakage detection. Effective privacy leakage detection methods in Android platform have been studied for a long time. Both static [16], [24] and dynamic [23] taint analysis techniques are developed and widely used to track private data. However, all these approaches only take into consideration fixed System APIs as sensitive data sources, like IMEI, phone number, etc. An exception is SUSI [35] that identifies more privacy sources in Android by using machine-learning to analyze Android system libraries. MudFlow [17] leverages such sources labelled by SUSI to mine apps for abnormal usage of sensitive data in mobile apps. However, these data are still walking around APIs and are mainly controlled by system. Further, UIPicker [32] and SUPOR [25] propose different approaches to identify sensitive data from app UIs, these approaches identify sensitive data from user input. UIPicker uses a SVM classifier to judge if a given element in a UI is privacy-critical or not, by learning only semantic features (e.g., if a set of privacy-related keywords appear simultaneously). In contrast, ClueFinder pipes the code structure as a feature to a SVM classifier to locate private data within app codes. These approaches mentioned above can not completely cover all private data identified by ClueFinder. BidText [26] introduces a bi-directional data propagation mechanism for detecting privacy leaks. Different from ours, BidText only detects whether a specific private data is leaked to system logs or network like HTTP requests, regardless of its responsibility. By comparison, our work focuses on the measurement against privacy leakage to third-party libraries, that is more helpful to the understanding of real world threats resulted from such privacy leakages. Similar to ClueFinder, Recon [36] detects the leakage of a wide range of users' private data, which is called personal identifiable information (PII) by Recon. However, different from ClueFinder in both approaches and purposes, Recon employs a dynamic analysis over mobile apps to directly confirm leaks by monitoring network traffic, while ClueFinder focuses on discovering private sources through its static analysis over decompiled app code. Also, Recon directly enables users to view PII leaks from network flows, while ClueFinder provides a basic tool for other existing approaches to detect more privacy leaks in a static way.

NLP analysis over mobile apps. There are lots of works utilizing NLP techniques to conduct semantic-based analysis against mobile apps for different purposes in the field of mobile security. Whyper [33] and AutoCog [34] inspect if a permission request is reasonable by analyzing its app descriptions. Similar to ClueFinder, they use dependency relation parsing to understand whether a given app contains descriptions about its permission usage. BidText [26] introduces dependency relation parsing to decide if a phrase or sentence is related to private data, however, it only excludes specific keywords with imperative negation (e.g., *"you should not"*) for labelling sensitive data. AsDroid [27] detects if a sensitive operation (e.g., sending SMS) matches its contents in the user interface, for identifying suspicious behaviors within apps. UIPicker [32] also utilizes some basic NLP techniques (e.g., stemming for keywords) as its pre-processing step for analysing textual resources in app UI for locating private information. However, both AsDroid and UIPicker did not consider dependency parsing over sensitive keywords within the sentence, thus may introduce false positives for recognizing privacy-related entities. All these approaches can further take advantages from ClueFinder, by employing a more comprehensive NLP analysis over app code or layout resources to improve their effectiveness.

VIII. Conclusion

In this paper, we give our research on detecting privacy leakage on mobile apps at a large-scale. To address the main challenge that many new types of private data (e.g., sensitive data on server-side) can not be effectively identified by traditional approaches, we propose ClueFinder, a new technique for sensitive data source discovery. ClueFinder leverages semantic information from app code, together with their unique program structures of their context to accurately and efficiently find privacy-related data within a given app. The evaluation results showed ClueFinder achieves a very high precision and outperforms similar existing work to a large extent. Also, using this technique, we investigated the potential information exposure to third-party libraries over 445,668 apps with a series of findings. These findings help better understand the privacy exposure risk and highlight the importance of data protection in today's software composition.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Chris Kanich for their insightful comments that helped improve the quality of the paper. We also thank Tongxin Li from Peking University, Nan Zhang from IU, and Li Tan for their assistance in our experiments. This work is funded in part by the National Program on Key Basic Research (NO. 2015CB358800), the National Natural Science Foundation of China (61602121, U1636204, 61602123), the Shanghai Sailing Program under Grant 16YF1400800. The IU author is supported in part by the NSF CNS-1527141, 1618493, ARO W911NF1610127 and Samsung gift fund.

References

[1] "Analytic sdk : co.inset.sdk," https://www.youtube.com/watch?v=sV0GwIl4oWs, accessed: 2017-08-10.

[2] "Bidtext-released version," https://bitbucket.org/hjjandy/toydroid.bidtext, accessed: 2017-08-10.

[3] "Google privacy policy," https://www.google.com/policies/privacy/, accessed: 2017-08-10.

[4] "Inmobi," http://inmobi.com, accessed: 2017-08-10.

[5] "Ironsec - user profiling function," http://www.ironsrc.com/atom/user-profiling/⟩, accessed: 2017-08-10.

[6] "Meet appboy - mobile engagement marketing tech startup," https://www.appboy.com/about/, accessed: 2017-08-10.

[7] "Mixpanel," https://mixpanel.com, accessed: 2017-08-10.

[8] "mopub," http://www.mopub.com/resources/docs/mopub-ui-account-setup/creating-managing-orders-and-line-items/line-item-targeting/⟩, accessed: 2017-08-10.

[9] "Proguard - the open source optimizer for java bytecode," https://www.guardsquare.com/en/proguard, accessed: 2017-08-10.

[10] "Requesting permissions," https://developer.android.com/training/permissions/requesting.html, accessed: 2017-08-10.

[11] "scikit-learn," http://scikit-learn.org/, accessed: 2017-08-10.

[12] "Sharesdk for android," http://www.mob.com/downloadDetail/ShareSDK/android, accessed: 2017-08-10.

[13] "Snaptee: T-shrit design," https://play.google.com/store/apps/details?id=co.snaptee.android, accessed: 2017-08-10.

[14] "The-paper-news," http://www.thepaper.cn/, accessed: 2017-08-10.

[15] "Tinder," https://play.google.com/store/apps/details?id=com.tinder, accessed: 2017-08-10.

[16] 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*, 2014.

[17] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 426–436.

[18] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 356–367.

[19] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th workshop on mobile computing systems and applications*. ACM, 2011, pp. 49–54.

[20] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies." in *USENIX Security Symposium*, 2013, pp. 131–146.

[21] Y.-W. Chen and C.-J. Lin, "Combining svms with various feature selection strategies." Springer, 2006, pp. 315–324.

[22] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for all! assessing user data exposure to advertising libraries on android," in *Proc. of NDSS'16*, 2016.

[23] 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 real-time privacy monitoring on smartphones," in *Communications of the ACM*, vol. 57, no. 3. ACM, 2014, pp. 99–106.

[24] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of android applications in droidsafe," in *Proc. of NDSS'15*, 2015.

[25] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "Supor: precise and scalable sensitive user input detection for android apps," in *24th USENIX Security Symposium*, 2015, pp. 977–992.

[26] J. Huang, X. Zhang, and L. Tan, "Detecting sensitive data disclosure via bi-directional text correlation analysis," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 169–180.

[27] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction." in *Proc. of ICSE'14*, 2014, pp. 1036–1046.

[28] Y. Z. X. Jiang and Z. Xuxian, "Detecting passive content leaks and pollution in android applications," in *Proc. of NDSS'13*, 2013.

[29] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*. Association for Computational Linguistics, 2003, pp. 423–430.

[30] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee, "The price of free: Privacy leakage in personalized mobile in-app ads," in *Proc. of NDSS'16*, 2016.

[31] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[32] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications," in *24th USENIX Security Symposium*, 2015, pp. 993–1008.

[33] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications." in *USENIX Security Symposium*, vol. 13, no. 20, 2013.

[34] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proc. of ACM CCS'14*, 2014, pp. 1354–1365.

[35] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proc. of NDSS'14*, 2014.

[36] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, "Recon: Revealing and controlling pii leaks in mobile network traffic," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2016, pp. 361–374.

[37] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard, "Covert communication in mobile applications (t)," in *Automated Software Engineering (ASE),30th IEEE/ACM International Conference*. IEEE, 2015, pp. 647–657.

[38] S. Son, D. Kim, and V. Shmatikov, "What mobile ads know about mobile users," in *Proc. of NDSS'16*, 2016.

[39] E. Steel, C. Locke, E. Cadman, and B. Freese, "How much is your personal data worth," 2013.

[40] 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)*, 2012, p. 10.

[41] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1017–1028.

[42] 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 *NDSS*, vol. 25, no. 4, 2012, pp. 50–52.

[43] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg, "Automated analysis of privacy requirements for mobile apps," in *Proc. of NDSS'17*, 2017.