Can We Trust the Privacy Policies of Android Apps?

Le Yu, Xiapu Luo [§], Xule Liu, Tao Zhang
Department of Computing, The Hong Kong Polytechnic University
The Hong Kong Polytechnic University Shenzhen Research Institute
{cslyu, csxluo, csxliu, cstzhang}@comp.polyu.edu.hk

***Abstract*—Recent years have witnessed the sharp increase of malicious apps that steal users' personal information. To address users' concerns about privacy risks, more and more apps are accompanied with privacy policies written in natural language because it is difficult for users to infer an app's behaviors according to the required permissions. However, little is known whether these privacy policies are trustworthy or not. It is worth noting that a questionable privacy policy may result from careless preparation by an app developer or intentional deception by an attacker. In this paper, we conduct the *first* systematic study on privacy policy by proposing a novel approach to automatically identify three kinds of problems in privacy policy. After tackling several challenging issues, we realize our approach in a system, named *PPChecker*, and evaluate it with real apps and privacy policies. The experimental results show that *PPChecker* can effectively identify questionable privacy policies with high precision. Moreover, applying *PPChecker* to 1,197 popular apps, we found that 282 apps (i.e., 23.6%) have at least one kind of problems. This study sheds light on the research of improving and regulating apps' privacy policies.**

I. Introduction

Smartphone has become an indispensable part of our daily lives with the great driving force from apps. Actually, the global app economy reached $53 billion in 2012 and expected to rise to $143 billion in 2016 [1]. Since the number of various malicious apps (e.g., malware, ransomware, adware, etc.) is also rapidly increasing [2], users are very concerned about the privacy risks introduced by apps [3], [4]. Although Android lists the permissions required by each app before installation, it is usually difficult for normal users to understand the potential threats by reading the permissions [5].

Alternatively, app developers can upload a privacy policy to Google Play store for declaring what information from users will be collected, used, retained, or disclosed [6]. A survey showed that 76% free apps in Google Play have provided privacy policies in 2012 [7]. Actually, many countries have enacted the privacy laws to force developers to add privacy policies, such as, California [8] and its California Online Privacy Protection Act(CalOPPA) [9], the Data Protection Directive(95/46/EC) [10] in European Union, etc. Federal Trade Commission (FTC) suggests mobile developers to prepare privacy policies for their apps [11] and provides guidance [12].

Unfortunately, it is not easy to prepare an accurate privacy policy for an app because of many reasons [13], [14]. For instance, it is not uncommon that the author of a privacy policy is not the developer of the app when the app is outsourced. As another example, if an app uses third-party libs, the app's privacy policy should cover these libs' behaviors or at least provide a pointer to their privacy policies. But Balebako et al. found that around half developers do not know for sure what information will be collected by third-party libs in their apps [15], because few third-party libs provide source code and they are often less transparent about data collection. It is worth noting that inaccurate privacy policies will lead to fines. For example, FTC fined *Path* $800,000 because its privacy policy failed to mention that it will retain users' information [16].

Therefore, an important question "Can we trust the privacy policies of Android Apps?" will be raised. Although manually dissecting apps and scrutinizing the privacy policies could answer this question, it is time-consuming and error-prone. In this paper, we propose a novel approach and develop the *first* system, named *PPChecker*, to automatically identify problems in privacy policy. It is challenging to design and develop *PPChecker* because of the following reasons. First, privacy policy is written in natural language, and the diversity of natural language makes it difficult to understand their meanings or extract useful information [17], [18]. Moreover, both the app's privacy policy and the third-party libs' privacy policies should be analyzed in order spot the inconsistency. Second, without assuming the availability of an app's source code, *PPChecker* should be able to understand an app's behaviors from its bytecode and contrast the behaviors with the information extracted from the privacy policy.

To tackle these challenging issues, *PPChecker* employs natural-language processing (NLP) techniques [19] to dissect privacy policies, and adopts program analysis approaches [20] to analyze apps (Section III). Moreover, we model the following three kinds of problems in privacy policies and propose algorithms to detect them (Section IV).

- **Incomplete privacy policy.** The privacy policy does not cover an app's all behaviors of accessing sensitive information, such as the case of `path` [16].
- **Incorrect privacy policy.** The privacy policy declares that the app will not access user information but the app does.
- **Inconsistent privacy policy**. The privacy policy of an app is in conflict with that of its third-party libs.

The output of *PPChecker* can help app companies to spot inaccuracies in their privacy policies, facilitate normal users to determine the trustworthiness of apps, and assist app market owners and organizations like FTC to identify questionable apps. It is worth noting that the inaccurate privacy policies can also be used to detect malicious apps. For example, the unrevealed behaviors in an incomplete privacy policy may come from the malicious component of a repackaged app [21]. Moreover, an adversary can create an incorrect privacy policy to fool users. In summary, our major contributions include:

[§]The corresponding author.

- To our best knowledge, this is the *first* investigation on automatically discovering problems in privacy policy for Android apps. We model three kinds of problems and design new algorithms to identify them.
- We propose and develop *PPChecker*, a novel system that adopts NLP and program analysis techniques, to automatically identify problems in privacy policy.
- We conduct careful evaluation on *PPChecker* by using real apps along with their privacy policies. The experimental results, which have been verified through manual checking, show that *PPChecker* can effectively detect those problems with high precision.

The rest of this paper is organized as follows. Section II defines the problem addressed by this paper and introduces necessary background knowledge. Section III details the design of *PPChecker* and Section IV elaborates on the new algorithms for detecting the problems in privacy policy, respectively. The experimental results are presented in Section V. We describe the limitations of *PPChecker* and possible solutions in Section VI. After introducing the related work in Section VII, we conclude the paper in Section VIII.

II. Background and Problem Definition

A. Privacy Policy

Privacy policy informs users what, when, why, and how information will be collected. For example, Fig. 1 shows a portion of the app (Golf Live Extra)'s privacy policy. It first says "*When you ..., we may collect and process ...*", indicating what and when information, including location, IP address, etc., will be collected. Then, the sentence "*we may share ... with ...*" informs readers which information will be shared with third parties. After that, it declares that the embedded third-party libs (e.g., Ad) will collect information. In this paper, we use *resource* and *information* interchangeably to denote the private data to be collected, used, retained or disclosed by an app as described in its privacy policy.

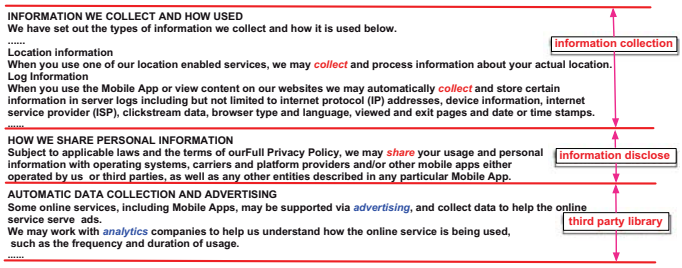

Fig. 1. App Golf Live Extra's privacy policy

B. Problem Definition

We aim at automatically identifying three kinds of issues in an app's privacy policy, including:

(1) Incomplete privacy policy. A well-written privacy policy should cover all privacy-related behaviors of the app. An incomplete privacy policy may result in a fine.

Fig.2 shows such an example where the description, permission and code snippet of the app `com.dooing.dooing` are listed. The description has a sentence "*Location aware tasks will help you to utilize your field force in optimum way.*" that indicates the use of location information. Moreover, the class `com.dooing.dooing.ee` calls location-related APIs including *getLatitude()* and *getLongitude()*. However, its privacy policy does not mention the behavior of collecting location information.

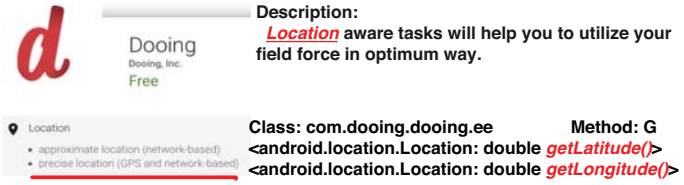

Fig. 2. Description and code snippet of `com.dooing.dooing`.

(2) Incorrect Privacy policy. A privacy policy should list an app's real behaviors. An incorrect privacy policy declares that the app will not collect, use, retain, or disclose personal information, but the app does.

For instance, the app *com.easyxapp.secret*'s privacy policy declares "*we will **not** store your real phone number , name and contacts*". However, we found from its bytecode that it obtains the contact information through the URI *<android.provider.ContactsContract$CommonDataKinds$Phone: android.net.Uri CONTENT_URI>* and writes it to log file.

(3) Inconsistent privacy policy. Since an app may integrate third-party libs, its privacy policy should cover the behaviors of third-party libs as a whole or point to their privacy policies. If an app's privacy policy declares that it will not access personal information but its third-party libs' privacy policies mention that they will conduct such behavior, we regard the app's privacy policy as an inconsistent privacy policy.

Fig.3 lists the information retrieved from the privacy policy of a very popular game app `com.imangi.templerun2` and that from the privacy policy of a third-party lib (i.e., `Unity3d`) used by this app. The solid line indicates the inconsistency between privacy policies. The app claims that it does *not* use/collect location information. However, the third-party lib declares that it will receive location information.

Fig. 3. Inconsistency in `com.imangi.templerun2`'s privacy policy.

III. PPChecker

We first give an overview of *PPChecker* in Section III-A and then detail its three major modules in Section III-B, Section III-C, and Section IV, individually. Tab. I lists the major symbols used in this paper.

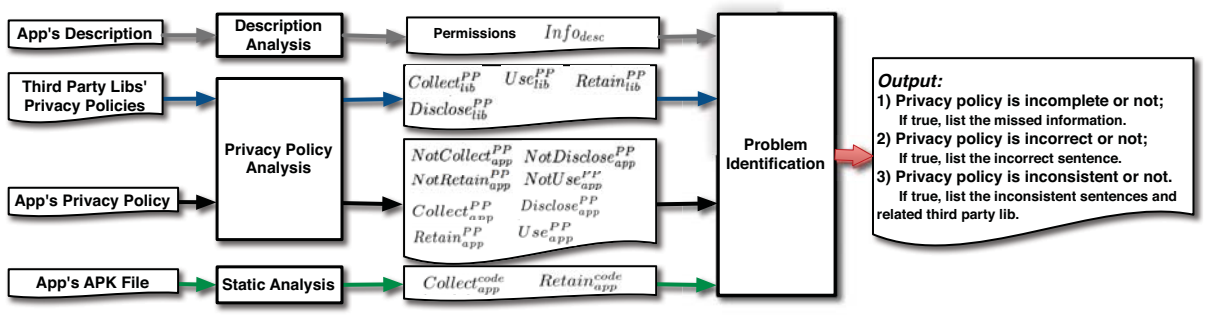

Fig. 4. Overview of *PPChecker*

A. System Overview

As shown in Fig. 4, *PPChecker* takes in an app's privacy policy, description, apk file, and third-party libs' privacy policies. The output includes: 1) whether the privacy policy is incomplete or not. If so, it lists the missed information; 2) whether the privacy policy is incorrect or not. If so, it enumerates the incorrect sentences; 3) whether the privacy policy is inconsistent or not. If so, it lists the inconsistent sentences and the relevant third-party lib's privacy policy. *PPChecker* consists of three major modules, including:

(1) Privacy policy analysis module (Section III-B). It analyzes a privacy policy to determine the information (not) to be collected, used, retained, or disclosed.

(2) Static analysis module (Section III-C). It inspects an app's bytecode to decide whether the app will collect or retain private information.

(3) Problem identification module (Section IV). It employs the models of three kinds of problems to identify incomplete privacy policy (IV-A), incorrect privacy policy (IV-B), and inconsistent privacy policy(IV-C).

B. Privacy Policy Analysis Module

1. Main Verbs, Sentences and Resources

There are four kinds of main verbs commonly used in privacy policy [22], [23], including:

• **Collect verbs**. They describe that one party accesses, collects, or acquires data from another party, such as "collect", "gather", etc. Let $VP_{collect}$ indicate such verbs.

• **Use verbs**. They depict that one party uses data from another party for certain purpose, such as "use", "process", etc. Let VP_{use} denote such verbs.

• **Retain verbs**. They mean that one party keeps the data collected from another party for a particular period of time or in a particular location, such as "retain", "store", etc. Let VP_{retain} represent such verbs.

• **Disclose verbs**. They indicate that one party transfers the collected data to another party, such as "disclose", "share", etc. Let $VP_{disclose}$ stand for such verbs.

Based on the four kinds of verbs, we define other symbols. Let $AppSent_*$ and $AppSents_*$ denote a $*$ type sentence and the set of such sentences in an app's privacy policy, respectively. The mark $*$ can be replaced by $collect$, use, $retain$, and $disclose$. Similarly, let $LibSent_*$ and $LibSents_*$ represent those sentences in a third-party lib's privacy policy.

By analyzing the sentences, *PPChecker* identifies the private information handled by main verbs. For positive sentences, we use $Collect_*^{PP}$, Use_*^{PP}, $Retain_*^{PP}$, and $Disclose_*^{PP}$ to denote the set of private information that will be collected, used, retained, and disclosed, respectively. The mark $*$ can be app for denoting the information mentioned in an app's privacy policy, or lib for indicating the information listed in a third-party lib's privacy policy.

A privacy policy may use negative sentences. For example, "*we cannot collect*" presents the opposite meaning of "*we can collect*". The former is negative sentence. We utilize $NotCollect_{app}^{PP}$, $NotUse_{app}^{PP}$, $NotRetain_{app}^{PP}$, and $NotDisclose_{app}^{PP}$ to denote the set of private information that will *not* be collected, used, retained, or disclosed, respectively, according to an app's privacy policy.

2. Steps in Privacy Policy Analysis

Fig. 5 shows the procedure of inspecting a privacy policy, which involves six steps detailed as follows.

Step 1: Sentence extraction. *PPChecker* extracts the content from a privacy policy and splits it into sentences. Following our system in [24], we use Beautiful Soup [25] to extract the content from each privacy policy in HTML format, and remove all non-ASCII symbols and some meaningless ASCII symbols. Note that we currently just consider privacy policies written in English and therefore the extracted content contains only English letters and some specified punctuation symbols.

Then, we use the natural language toolkit (NLTK) [26] to divide the text into sentences. Since NLTK divides an enumeration list into individual sentences, it may cause errors. For example, the sentence "*we will collect the following information: your name; your IP address; your device ID.*" will be divided into four parts, and the three resources after ":" are regarded as three sentences. To address this issue, *PPChecker* checks the sequence of sentences from NLTK one by one. If the previous sentence ends with symbol ":" or ";"

TABLE I. MAJOR SYMBOLS IN THIS PAPER

Symbol	Meaning
$Sent_{collect}$	a sentence whose main verb $\in VP_{collect}$
$Sent_{use}$	a sentence whose main verb $\in VP_{use}$
$Sent_{retain}$	a sentence whose main verb $\in VP_{retain}$
$Sent_{disclose}$	a sentence whose main verb $\in VP_{disclose}$
$AppSent_{collect}$	a sentence in app's privacy policy whose main verb $\in VP_{collect}$
$AppSent_{use}$	a sentence in app's privacy policy whose main verb $\in VP_{use}$
$AppSent_{retain}$	a sentence in app's privacy policy whose main verb $\in VP_{retain}$
$AppSent_{disclose}$	a sentence in app's privacy policy whose main verb $\in VP_{disclose}$
$LibSent_{collect}$	a sentence in lib's privacy policy whose main verb $\in VP_{collect}$
$LibSent_{use}$	a sentence in lib's privacy policy whose main verb $\in VP_{use}$
$LibSent_{retain}$	a sentence in lib's privacy policy whose main verb $\in VP_{retain}$
$LibSent_{disclose}$	a sentence in lib's privacy policy whose main verb $\in VP_{disclose}$
$Collect_{app}^{PP}$	resources to be collected according to an app's privacy policy
Use_{app}^{PP}	resources to be used according to an app's privacy policy
$Retain_{app}^{PP}$	resources to be retained according to an app's privacy policy
$Disclose_{app}^{PP}$	resources to be disclosed according to an app's privacy policy
$Collect_{lib}^{PP}$	resources to be collected according to a lib's privacy policy
Use_{lib}^{PP}	resources to be used according to a lib's privacy policy
$Retain_{lib}^{PP}$	resources to be retained according to a lib's privacy policy
$Disclose_{lib}^{PP}$	resources to be disclosed according to a lib's privacy policy
$NotCollect_{app}^{PP}$	resources that will not be collected according to an app's privacy policy
$NotUse_{app}^{PP}$	resources that will not be used according to an app's privacy policy
$NotRetain_{app}^{PP}$	resources that will not be retained according to an app's privacy policy
$NotDisclose_{app}^{PP}$	resources that will not be disclosed according to an app's privacy policy
$Collect_{app}^{code}$	resources collected by an app according to its code
$Retain_{app}^{code}$	resources retained by an app according to code
$Info_{desc}$	the set of resources used by an app according to its description

or lowercase letters, *PPChecker* appends the current sentence to the previous one. Finally, *PPChecker* turns all letters into lower case.

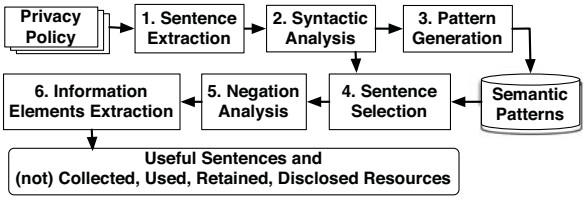

Fig. 5. The procedure of analyzing a privacy policy.

Step 2: Syntactic analysis. It parses sentences and obtains syntactic information. For each sentence, we use Stanford Parser [27] to obtain its syntactic tree and dependency relations. For example, Fig. 6 shows the syntactic information of the sentence: *"we will provide your information to third party companies to improve service"*. The left part is the parse tree

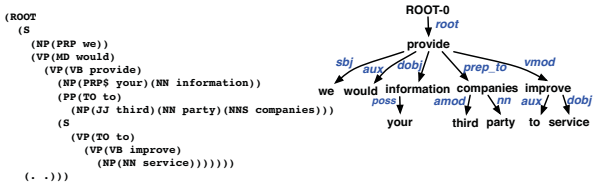

Fig. 6. Syntactic information of sentence: "*we would provide your information to third party companies to improve service*".

structure and the right part is the typed dependency relation.

The parse tree breaks a sentence into phrases and shows them in a hierarchy structure, where each phrase occupies one line. The parse tree also contains the *part-of-speech* (POS) tags of words and phrases, which are assigned based on a word's or a phrase's syntax behavior. Common POS tags include noun (NN), verb (VB), adjective (ADJ), adverb (ADV), pronoun (PRP), etc. In Fig. 6, "*provide*" is a verb, and we can find its object noun phrases "*your information*" in its sub-tree.

The typed dependency describes the relation between words. Common relations include: sbj that means the subject, $dobj$ that represents direct object, $root$ that stands for the relation point to the root word of a sentence, $nsubjpass$ that refers to a noun phrase being the syntactic subject of a passive verb. $prep\_to$ means a verb, adjective, or noun modified by a prepositional starting with "to" (*e.g.*, "go to store"). $auxpass$ means passive auxiliary [28]. In Fig. 6, "*provide*" is the root word of this sentence. The subject of this sentence is "*we*", and the object is "*your information*".

The syntactic information is used in the following pattern generation step and the sentence selection step.

Step 3: Pattern generation. Existing privacy policy analysis systems (*e.g.*, [18] [29]) use pre-defined patterns to find sentences relevant to information collection. We enhance the bootstrapping mechanism [30] to *automatically* find patterns from privacy policies according to a simple seed pattern. We will use the example in Fig.7 to explain the bootstrapping mechanism and then describe the enhancement.

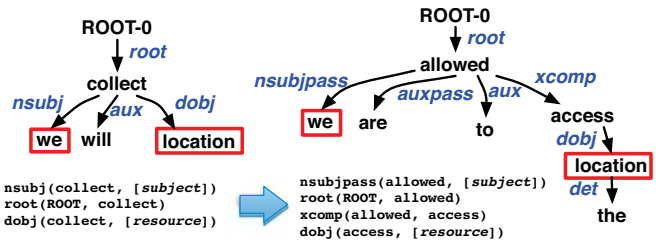

Fig. 7. Example of finding a new pattern.

We first prepare a corpus containing sentences relevant to information collection, utilization, reservation, and disclosure. The seed pattern is *subject-verb-object* and the initial verbs include "collect", "use", "retain", and "disclose". Obviously, the seed pattern matches the left sentence "*we will collect location*" in Fig.7. We collect the subjects and the objects of all matched sentences (e.g., "we" and "location" in the left sentence), and insert the subjects and the objects with

frequencies higher than the median into the subject list and the object list, respectively.

Then, we look for new patterns by matching the subject and the object in each typed dependency tree with the elements in the subject list and the object list. For the right sentence in Fig.7, since its typed dependency tree contains the subject "we" and the object "location", we extract the shortest path between them as the new pattern. Therefore, the new pattern *subject-"allowed"-"access"-object* is extracted. All new patterns are inserted into the pattern list for next iteration, and the algorithm stops when no new pattern is found.

We enhance the algorithm in [30] from two aspects. One is how to handle semantic drift, which refers to the deviation of the meaning of new terms from that of the seed patterns [31]. We propose using three blacklists to address this issue. First, since we only focus on an app's behaviors, *PPChecker* maintains a subject blacklist, which contains words such as "you", "user", "visitor", to remove the sentences describing the app's users. Second, since we only care about four kinds of behaviors of an app, *PPChecker* removes verbs unrelated to such behaviors (e.g., "have", "make", etc.). Third, *PPChecker* discards sentences that are irrelevant to personal information by employing a blacklist for objects (e.g., services, etc.).

The other one is how to rank and acquire new patterns. After finding new patterns from the corpus, we rank them for selecting important ones. Given a pattern p, it is important if it can match more sentences about information collection, usage, retention, or disclose. However, p's importance decreases if it can match many irrelevant sentences. To score each pattern, we construct two sentence sets from real policy policies. One is called positive sentence set that contains sentences about information collection, usage, retention, or disclose, whereas the other one, named negative sentence set, includes unrelated sentences. For a pattern p, we use $pos(p)$ to denote the number of positive sentences p can match, and utilize $neg(p)$ to represent the number of negative sentences p can match. Let $unk(p)$ indicate the number of sentences that cannot be matched by any pattern. Then, the $accuracy$ and $confidency$ of pattern p are defined as:

$$acc(p) = \frac{pos(p)}{pos(p) + neg(p)}, conf(p) = \frac{pos(p) - neg(p)}{pos(p) + neg(p) + unk(p)} \quad (1)$$

Moreover, the score of p equals $Score(p) = conf(p) * log|pos(p)|$. The top n patterns will be used in the following sentence selection step.

Step 4: Sentence selection.

PPChecker uses the generated patterns to identify sentences from privacy policies. The matched sentences are regarded as *useful sentences* and others will be discarded. We use five patterns listed in Tab. II as an example to illustrate this procedure. P1 is the seed pattern. P2 is the passive voice version of P1. The other three patterns (*i.e.*, P3, P4, and P5) are automatically extracted from the corpus.

To find sentences matching patterns P1 and P2, we parse each sentence's typed dependency to extract its root word, and then check whether its root word belongs to the four main verb categories. If so, we keep the sentence as a useful sentence. Moreover, if the useful sentence's root verb has $auxpass$ relation with other words, this sentence has passive voice (*i.e.*, P2). Otherwise, it has active voice (*i.e.*, P1).

The pattern P3 describes that the subject is allowed to do something. To match it, the root word of a sentence should be "allowed". Moreover, its "allowed" should have an $auxpass$ relation with another word and an $xcomp$ relation with a verb. The verb in the infinitive phrase should belong to the main verb categories. The pattern P4 represents that the subject is able to do something. To match it, the root word of a sentence should be "able". Moreover, "able" is modified by an open clausal complement and should have an $xcomp$ relation with another verb that belongs to the main verb categories. The pattern P5 is an adverbial clause. To match it, the root word of a sentence should belong to the main verb categories and have an $advcl$ relation with another verb.

Step 5: Negation analysis. *PPChecker* determines whether a sentence is negative by checking the existence of negation words in two places [17]. One is the subject for identifying sentences like "*nothing will be collected*". The other refers to the words used to modify the root word, such as "*we will not collect information*". We adopt the negation word list from [32], because it includes the negative verbs (e.g., "prevent"), negative adverbs (e.g., "hardly"), negative adjectives (e.g., "unable"), and negative determiners (e.g., "no"). **Step 6:**

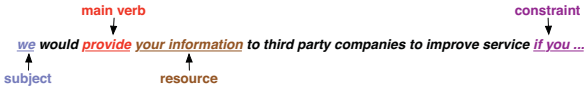

Fig. 8. Information elements in sentence: *"we will provide your information to third party companies to improve service if you ..."*

Information elements extraction. From each useful sentence, we look for four elements, including main verb, action executor, resource, and constraint. For example, Fig. 8 shows the information elements in the sentence: *"we will provide your information to third party companies to improve service if you ..."*, where the main verb is "provide", the subject is "we", its object is "your information", and its constraint is "if you ...". The subject, main verb, and resource are used in the problem identification module (Section IV-A). The constraint is used to identify and remove some specific sentences, including (1) when a user registers an account through a website, although the website may collect private information, *PPChecker* does not consider it; (2) when a user visits the website that will log the IP address and other information, since the behavior is not performed by the app, *PPChecker* ignores it.

The main verb is the key verb of a sentence. In the typed dependency relation, the main verb is the word that has $root$ relation with a virtual "ROOT-0" word, such as "provide" in Fig. 6. The action executor is the entity who conducts the main verb. In the typed dependency relation, it is the word that has sbj relation with main verb, such as "we" in Fig. 6.

The resource is the data used by action executor, such as "information" shown in Fig. 6. If the sentence is active voice, the resource has $dobj$ relation with the main verb. Otherwise, the resource is the subject that has $nsubjpass$ relation with main verb. For instance, in the sample sentence "*your location will be collected by us*", the resource "your personal information" is the subject of this sentence.

Two kinds of constraints are extracted: pre-condition and post-condition. Pre-condition starts with "if", "upon", "un-

TABLE II. SAMPLE PATTERNS FOR SELECTING SENTENCES

#	Pattern Name	Semantic Pattern	Sample Sentence
P1	Active Voice	$sbj\ VP_*\ resource$	We will use your personal information.
P2	Passive Voice	$resource\ VP_*^{passive}$	Your personal information will be used.
P3	Passive Allow Expression	sbj "be allowed to" $VP_*\ resource$	We are allowed to access your personal information.
P4	Ability Expression	sbj " able to" $VP_*\ resource$	We are able to collect location information.
P5	Purpose Expression	$sbj\ VP_*\ resource$ to $VP_*\ resource$	We use GPS to get your location.

less", post-condition starts with "when", "before". We collect the constraints by extracting the sub-tree that starts with these words from the syntactic tree.

C. Static Analysis Module

Given an app, *PPChecker* conducts static code analysis on its *dex* file to determine the following information: (1) private information collected by the app (i.e., $Collect_{app}^{code}$) and (2) private information retained by the app (i.e., $Retain_{app}^{code}$).

1. Conducting the Static Analysis

We develop *PPChecker*'s static analysis module based on our static analysis framework, *VulHunter* [33], and improves it from several aspects. Given an app, *PPChecker* extracts the `AndroidManifest.xml` and the `dex` file from the `APK` file. If the app is packed, we use our unpacking tool DexHunter [34] to recover the `dex` file. By parsing the `AndroidManifest.xml` file and the `dex` file, *PPChecker* constructs an Android property graph (APG) [33] that integrates abstract syntax tree (AST), interprocedure control-flow graph (ICFG), method call graph (MCG), and system dependency graph (SDG) of the app, and stores it into a graph database. By doing so, *PPChecker* can determine the collected and retaining information by performing queries.

To enhance the accuracy of static analysis, we employ IccTA [35] to identify the source and the target of an intent, and utilize EdgeMiner [36] to determine the implicit callbacks (e.g., from *setOnClickListener()* to *onClick()*). Moreover, to improve the performance of the static taint analysis, we also include the source-sink paths identified by FlowDroid [37] in the graph database. The sources refer to sensitive APIs to be described in the next sub-section. The sinks refer to APIs that store information into a log (e.g., *Log.d()*) or a file (e.g., *FileOutputStream.write()*), or send it out through network (e.g., *AndroidHttpClient.execute()*), SMS(e.g., *sendTextMessage()*), or bluetooth (e.g., *BluetoothOutputStream.write()*).

2. Identifying the Collected Information

An app can collect personal information through two approaches. One is to invoke sensitive APIs, such as calling *getDeviceId()* for obtaining device ID. The other one is to get the information through content provider [38], such as calling *android.content.ContentResolver.query()* with *content://com.android.calendar* to access calendar information. We regard invoking such a query function with specific URIs as calling a sensitive API. We will describe how *PPChecker* handles them in the following paragraphs, individually.

PPChecker looks for 68 sensitive APIs covering the information about device ID, IP address, cookie, location, account, contact, account, calendar, telephone number, camera, audio, and app list. Such information is commonly listed in privacy policy. We select these APIs from the data sets in [38], [39].

We select 12 URI strings along with 615 URI fields from the data set in [38]. *PPChecker* conducts the following steps to locate the statements calling the query function and determine the corresponding URIs [40]. First, *PPChecker* locates the statements that access the content provider. Then, it identifies all paths starting from each query statement by querying the graph database, and collect all statements on the paths. After that, it inspects these statements' parameters, and records used URIs. For example, if a statement calls *Uri.parse(content://com.android.calendar)* to get URI object and the returned URI object is used in *ContentResolver.query()*, we record the parameter *content://com.android.calendar*.

Since some sensitive APIs may not be called by the app (e.g., infeasible code), we conduct the reachability analysis from the app's entry points, including life-cycle callbacks (e.g., *Activity.onCreate()*), major components' entry functions (e.g., *query()* in content provider), and UI related callbacks (e.g., *onClick()*), to the invocation of each sensitive API by querying the graph database. We do not consider those sensitive APIs to which there are not feasible paths from entry points. For each remaining sensitive API, we check their class names. If the class name has the same prefix as the package name of the app, we regard the app as the caller of this API. To determine the collected information, we map the sensitive

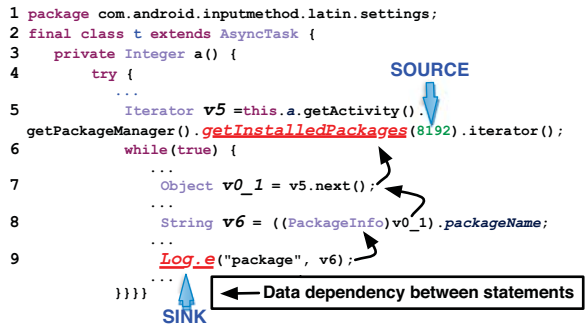

Fig. 9. Code snippet of `com.qisiemoji.inputmethod`: get installed package list and write it to log

APIs and the URI strings to private information by analyzing their official documents. For example, the API *getDeviceId()* is mapped to "device ID" and the URI string *content://contacts* is mapped to "contact". For the URI fields, since *PScout* provides the map between URI fields and permissions [38], we map these fields to the private information according to the corresponding permissions. For instance, since *PScout* maps "*<android.provider.Telephony$Sms: android.net.Uri CONTENT_URI>*" to permission `android.permission.RECEIVE_SMS`, we map this URI field to "SMS".

3. Determining the Retained Information

We perform static taint analysis to determine the retained information. More precisely, if there is an execution path connecting a source to a sink, the sensitive information relevant to the source is retained. For instance, Fig.9 shows a code snippet of `com.qisiemoji.inputmethod`. *PPChecker* traverses from the sensitive API *getInstalledPackages()* (Line 5), which retrieves the list of installed packages device, and ends at a sink function *Log.e()* (Line 9). By using the map between API/URI strings(fields) and private information, *PPChecker* obtains the information retained by the app.

D. Description Analysis Module

We use the state-of-art description analysis system, *AutoCog*, to map an app's description to permissions [41]. Then, we map the permissions to private information by analyzing the official document. For example, permission `ACCESS_FINE_LOCATION` is mapped to "location", "latitude", "longitude". Let $Info_{desc}$ denote the collected information that is inferred from the app's description.

IV. Problem Identification Module

A. Detecting Incomplete Privacy Policy

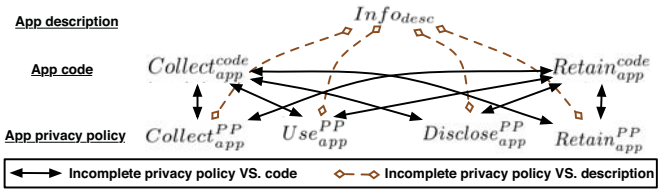

Fig. 10. Model of incomplete privacy policy.

Fig.10 illustrates that the incomplete privacy policy can be identified by contrasting its information with the app's description or bytecode. First, if the information listed in a privacy policy cannot cover that inferred from the description, we can detect an incomplete privacy policy. Second, if the information mentioned in a privacy policy cannot cover the collected or retained information determined from bytecode, we can identify an incomplete privacy policy.

Detecting incomplete privacy policy through description. The description analysis module provides the information to be used by an app (i.e., $Info_{desc}$) while the privacy policy analysis module lists the information to be used/collected/retained/disclosed by the app (line 1, $PPInfos$ in Alg. 1). We compare each information in $Info_{desc}$ with all the information identified from the privacy policy(line 5-9 in Alg. 1). If no private information pairs can be matched successfully, then the $Info$ is missed by the privacy policy (saved in line 10-12 in Alg. 1), and the privacy policy is an incomplete one (return the missed information in line 14-16 in Alg. 1).

Here, "matching" means that two kinds of information refer to the same thing. To measure the semantic similarity of two kinds of information, we use Explicit Semantic Analysis (ESA) [42]. Given two texts, ESA first maps each of them to a vector representation by using a knowledge base. Then, ESA calculates the similarity of these two vectors to get the semantic similarity of these two texts. If the similarity reaches a threshold, we regard them as the same thing (line 5 in Algorithm 1). Currently, we adopt 0.67 as the threshold following [41].

Algorithm 1: Detect Incomplete Privacy Policy through Description

Input: $Collect_{app}^{PP}, Use_{app}^{PP}, Retain_{app}^{PP}, Disclose_{app}^{PP}$: information collected, used, retained or disclosed by app privacy policy; $Info_{Desc}$: information that app's description says will use.
Output: $ProblemInfos$: Return the missed information if the privacy policy is incomplete; Null: Return null if the privacy policy is not incomplete.

```
1  PPInfos = Collect_app^PP ∪ Use_app^PP ∪ Retain_app^PP ∪ Disclose_app^PP;
2  ProblemInfos = [];
3  for Info in Info_desc do
4      FindSimilarInfo = 0;
5      for PPInfo in PPInfos do
6          if Similarity(Info, PPInfo) > threshold then
7              FindSimilarInfo = 1;
8          end
9      end
10     if FindSimilarInfo == 0 then
11         ProblemInfos.append(Info); //Save the missed Info
12     end
13 end
14 if ProblemInfos.length() > 0 then
15     return ProblemInfos; //Privacy policy is incomplete
16 end
17 return Null; //Privacy policy is not incomplete
```

Detecting incomplete privacy policy through code.

The code analysis module provides the information collected or retained by analyzing an app's bytecode. We compare each information with all the information identified from the privacy policy. If the privacy policy does not cover such information, an incomplete privacy policy is found. The algorithm is shown in Algorithm 2. Note that some information requires permission (*e.g.*, location requires `ACCESS_COARSE_LOCATION`). In this case, we only consider the app that requires the corresponding permissions.

Algorithm 2: Detect Incomplete Privacy Policy through Code

Input: $Collect_{app}^{PP}, Use_{app}^{PP}, Retain_{app}^{PP}, Disclose_{app}^{PP}$: information collected, used, retained or disclosed by app privacy policy; $Collect_{app}^{code}$: information collected in app code. $Retain_{app}^{code}$: information retained in app code
Output: $ProblemInfos$: Return the missed information if the privacy policy is incomplete; Null: Return null if the privacy policy is not incomplete.

```
1  PPInfos = Collect_app^PP ∪ Use_app^PP ∪ Retain_app^PP ∪ Disclose_app^PP;
2  ProblemInfos = [];
3  for CodeInfo in (Collect_app^code ∪ Retain_app^code ) do
4      FindSimilarInfo = 0;
5      for PPInfo in PPInfos do
6          if Similarity(CodeInfo, PPInfo) > threshold then
7              FindSimilarInfo = 1;
8          end
9      end
10     if FindSimilarInfo == 0 then
11         ProblemInfos.append(CodeInfo); // save the missed CodeInfo
12     end
13 end
14 if ProblemInfos.length() > 0 then
15     return ProblemInfos; //Privacy policy is incomplete
16 end
17 return Null; //Privacy policy is not incomplete
```

B. Discovering Incorrect Privacy Policy

Fig.11 shows that the incorrect privacy policy can be identified by contrasting its information with the app's description or bytecode. First, if the information mentioned in a privacy

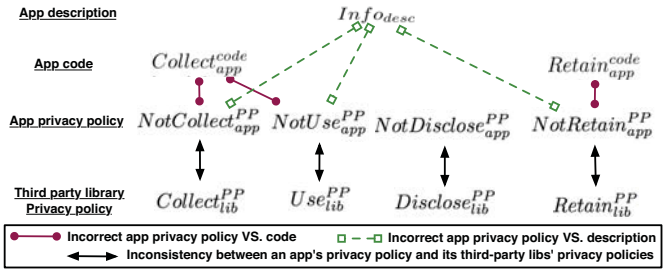

Fig. 11. Model of incorrect and inconsistent privacy policy.

policy is in conflict with that inferred from the description, we can detect an incorrect privacy policy. Second, if the information listed in a privacy policy conflict with the collected or retained information determined from bytecode, we can identify an incorrect privacy policy.

It is worth noting that the semantic difference between the collected resources and the used resources lies in whether the privacy policy declares the purpose of obtaining such resources. Since the static code analysis cannot infer such purpose from bytecode, we contrast both $NotCollect^{PP}_{app}$ and $\cup NotUse^{PP}_{app}$ with $Collect^{code}_{app}$.

Discovering incorrect privacy policy through description. Algorithm 3 shows how to discover incorrect privacy policy through description. Similar to Algorithm 1, for each information in $Info_{desc}$, we compare it with all information that will *not* be collected, used, or retained according to the privacy policy (line 5-9 in Algorithm 3). If there is at least a pair of similar information, the privacy policy is incorrect (line 10-12 in Algorithm 3).

Algorithm 3: Discover Incorrect Privacy Policy through Description

Input: $NotCollect^{PP}_{app}, NotUse^{PP}_{app}, NotRetain^{PP}_{app}$: information not collected, used, or retained by app privacy policy; $Info_{desc}$: information that app's description says to use.
Output: $ProblemInfos$: Return the problem information if the privacy policy is violative; Null: Return null if the privacy policy is correct.

1 $PPInfos = NotCollect^{PP}_{app} \cup NotUse^{PP}_{app} \cup NotRetain^{PP}_{app}$;
2 $ProblemInfos$ = [];
3 **for** $Info$ *in* $Info_{desc}$ **do**
4 $FindSimilarInfo$ = 0;
5 **for** *PPInfo in PPInfos* **do**
6 **if** *Similarity(Info, PPInfo) > threshold* **then**
7 $FindSimilarInfo$ = 1;
8 **end**
9 **end**
10 **if** $FindSimilarInfo$ == *1* **then**
11 $ProblemInfos$.append($Info$); //Save the problem information
12 **end**
13 **end**
14 **if** $ProblemInfos.length() > 0$ **then**
15 return $ProblemInfos$; //Privacy policy is incorrect
16 **end**
17 return Null; //Privacy policy is correct

Discovering incorrect privacy policy through code. *PPChecker* uses Algorithm 4 to detect incorrect privacy policy that declares not to retain certain information but the app does in code. Moreover, it will detect incorrect privacy policy that declares not to collect or use certain information but the app does. The corresponding algorithm is similar to Algorithm 4. The only difference is that we use $NotCollect^{PP}_{app} \cap NotUse^{PP}_{app}$ to replace $NotRetain^{PP}_{app}$ and utilize $Collect^{code}_{app}$ to replace $Retain^{code}_{app}$.

Algorithm 4: Discover Incorrect Privacy Policy through Code

Input: $NotRetain^{PP}_{app}$: information not retained by app privacy policy; $Retain^{code}_{App}$: information retained in app code
Output: $ProblemInfos$: Return the problem information if the privacy policy is incorrect; Null: Return null if the privacy policy is correct.

1 $ProblemInfos$ = [];
2 **for** *CodeInfo in* $Retain^{code}_{app}$ **do**
3 $FindSimilarInfo$ = 0;
4 **for** *PPInfo in* $NotRetain^{PP}_{app}$ **do**
5 **if** *Similarity(CodeInfo, PPInfo) > threshold* **then**
6 $FindSimilarInfo$ = 1;
7 **end**
8 **end**
9 **if** $FindSimilarInfo$ == *1* **then**
10 $ProblemInfos$.append($CodeInfo$);
11 //Save the problem information
12 **end**
13 **end**
14 **if** $ProblemInfos.length() > 0$ **then**
15 return $ProblemInfos$; //Privacy policy is incorrect
16 **end**
17 return Null; //Privacy policy is correct

C. Revealing Inconsistent Privacy Policy

Algorithm 5: Reveal Inconsistency between an app's privacy policy and its third-party libs' privacy policies

Input: PP_{App}: app privacy policy, PP_{Lib}: third party library privacy policy
Output: $ProblemSents$: Return the inconsistent sentences if the privacy policy is inconsistent; Null: Return null if the privacy policy is not inconsistent.

1 $ProblemSents$ = [];
2 **for** *i in range (1, m)* **do**
3 **for** *j in range (1, n)* **do**
4 $VPCate_{app} = getVerbCategory(AppSent_i)$;
$VPCate_{lib} = getVerbCategory(LibSent_j)$;
5 **if** $(VPCate_{app} == VPCate_{app}) \wedge$ $(!IsPositive(AppSent_i)) \wedge (IsPositive(LibSent_j))$ **then**
6 $AppResSet = getRes(AppSent_i)$;
7 $LibResSet = getRes(LibSent_j)$;
8 **for** $AppRes$ *in* $AppResSet$ **do**
9 **for** $LibRes$ *in* $LibResSet$ **do**
10 **if** *Similarity(AppRes,LibRes) > threshold* **then**
11 $ProblemSents$.add($[AppSent_i, LibSent_j]$);
//Save the inconsistent sentences
12 **end**
13 **end**
14 **end**
15 **end**
16 **end**
17 **end**
18 **if** $ProblemSents.length()>0$ **then**
19 return $ProblemSents$; //Privacy policy is inconsistent
20 **end**
21 return Null; //Privacy policy is not inconsistent

Fig.11 also depicts how to determine the inconsistency between an app's privacy policy and its third-party libs' privacy policies. To identify the third-party libs used in app, we maintain a list of class name prefixes of third-party libs. Then, the static analysis module goes through all class names to find the third-party libs integrated in the app. Given an app with m useful sentences in its privacy policy and n useful sentences in its third-party libs' privacy policies, we compare $AppSent_i$ $(1 \leq i \leq m)$ with $LibSent_j$ $(1 \leq j \leq n)$. More precisely, given $AppSent_i$ and $LibSent_j$, if the following requirements are satisfied, then there exists an inconsistency.

(1) $AppSent_i$'s and $LibSent_j$'s main verbs belong to the same category (i.e., $VP_{collect}, VP_{use}$, VP_{retain}, or $VP_{disclose}$);
(2) $AppSent_i$ is a negative sentence and $LibSent_j$ is a positive sentence;
(3) $AppSent_i$ and $LibSent_j$ refer to the same resource.

The detailed steps are shown in Algorithm 5. Note that line 4-5 are related to requirement (1) and (2): Function $getMainVerbCategory()$ returns the category of a sentence's main verb; Function $IsPositive()$ returns true if a sentence is positive; Function $getRes()$ returns the resources extracted from a sentence. line 6-14 are related to requirement (3) and the function $Similarity()$ decides whether two resources refer to the same resource by using ESA [42].

Disclaimer in privacy policy. Some app privacy policies declare that they are not responsible for behaviors of third parties. For example, app `com.shortbreakstudios.HammerTime` declares "*We encourage you to review the privacy practices of these third parties before disclosing any personally identifiable information, as we are not responsible for the privacy practices of those sites*" in its privacy policy. In this case, if the app privacy policy is inconsistent with third party lib privacy policy, such inconsistency is ignored.

V. Experimental Result

A. Data Set

We downloaded top 500 apps from each category in Goolge Play store and then randomly selected 1,197 apps that have descriptions and privacy policies in English. We examined the privacy policies of three kinds of third-party libs, including:
(1) Ad libs. 52 out of top 90 popular Ad libs in [43] were selected because they have privacy policies in English.
(2) Social libs. 9 out of 18 most popular social network libs in [44] were chosen because they offer privacy policies in English.
(3) Development tools. We picked 20 most commonly used development tools with privacy policies in English from [45] because the majority of other tools do not have websites showing their privacy policies.
We found that 879 apps (73%) under examination employ at least one of the above third-party libs.

B. Pattern Selection

The number of selected patterns (i.e., n) affects the performance of *PPChecker*. If a useful sentence cannot be matched by any selected patterns, then a false negative (FN) is generated. If an irrelevant sentence is matched by one selected pattern, then a false positive (FP) is raised.

To select a proper number of patterns, we created a positive sentence set and a negative sentence set, each of which contains 250 sentences selected from 100 privacy policies. All of them have been verified manually. Fig.12 shows the false positive rate and the false negative rate when n increases. We set n to the value that leads to the minimal summation of the false positive rate and the false negative rate. Therefore, n is set to 230 with detecting rate 88.0% (i.e., false negative rate is 12%) and false positive rate 2.8%. Note that users can adopt other strategies (e.g., constant false alarm rate) to select n.

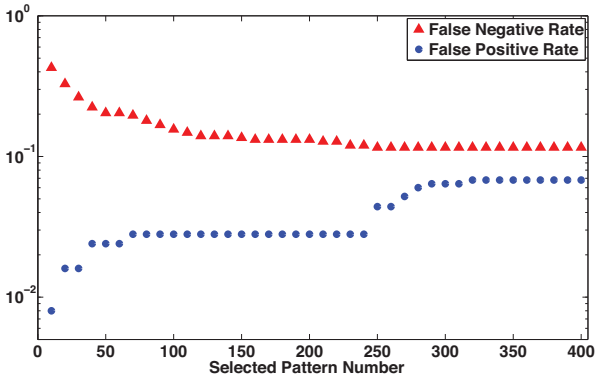

Fig. 12. False positive rate and false negative rate with different n.

C. Detecting Incomplete Privacy Policy

Detecting incomplete privacy policy through description. Contrasting an app's description and its privacy policy, *PPChecker* found 64 questionable apps. Tab. III lists the permissions that lead to the incompleteness and the corresponding number of suspicious apps. In other words, these permissions can be inferred from those apps' descriptions whereas their privacy policies do not cover them. We can see that the location related permissions (i.e.,`ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`) affect more apps. Many of these apps belong to the category of weather and the category of map, whose apps need the location information to provide services. Moreover, the permissions `READ_CONTACTS` and `GET_ACCOUNTS` also affect many apps.

Permission	Num. of Questionable apps
ACCESS_COARSE_LOCATION	14
ACCESS_FINE_LOCATION	19
CAMERA	6
GET_ACCOUNTS	11
READ_CALENDAR	2
READ_CONTACTS	12
WRITE_CONTACTS	1

TABLE III. Permissions leading to incomplete privacy policy and the number of corresponding apps.

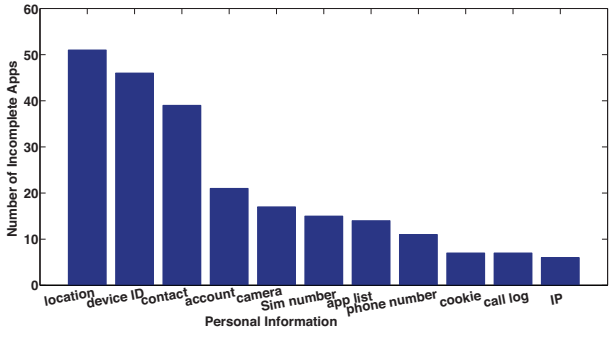

Fig. 13. Distribution of private information collected or retained by apps with incomplete privacy policies.

Detecting incomplete privacy policy through code. By analyzing apps' bytecode, *PPChecker* found 195 incomplete privacy policies. After manually checking the code and the

privacy policy of these apps, we found that 180 apps do have the problem whereas 15 apps are false positives.

Within these 180 incomplete privacy policies, *PPChecker* found 234 records of missed information, among which 32 records of missed information are retained. Fig. 13 lists the distribution of missed information. We can observe that the location information is the most common information missed in incomplete privacy policies. This result is consistent with the result described in Tab.III. We also checked the corresponding APIs used by those apps with incomplete privacy policy and found that for the location, *getLongitude()*, *getLatitude()*, and *getLastKnownLocation(java.lang.String)* are the three most commonly invoked APIs.

False positives. After inspecting the result, we found that the false positives are caused by errors in extracting private information from some sentences. For example, for the sentence "*in addition to your device identifiers, we may also collect: the name you have associated with your device*", we only extracted "name" since it is the object of the action "collect", but failed to extract "device identifier". As mentioned in Section VI, we will deal with such constraints in future work.

False negatives. Since checking false negatives requires a lot of manual effort, we randomly selected 20 apps to determine whether *PPChecker* results in false negatives. The result shows that *PPChecker* identifies all incomplete privacy policies.

D. Discovering Incorrect Privacy Policy

Discovering incorrect privacy policy through description. *PPChecker* found 2 suspicious apps: "com.marcow.birthdaylist" and "com.herman.ringtone". For example, the privacy policy of "com.marcow.birthdaylist" says *"we are **not** collecting your data of birth, phone number, name or other personal information, nor those of your contact."*. But its description describes the use of contact, *"This app synchronizes all birthdays with your contacts list and facebook."*

Discovering incorrect privacy policy through code. *PPChecker* found the same two apps with incorrect privacy policies by comparing $NotCollect_{app}^{PP}$ and $Collect_{app}^{code}$. Both "com.marcow.birthdaylist" and "com.herman.ringtone" declare not to collect contact information in their privacy policies, but they do query the corresponding URIs in the code.

By comparing $NotRetain_{app}^{PP}$ and $Retain_{app}^{code}$, *PPChecker* found another two apps with incorrect privacy policies. One is "com.easyxapp.secret". Its privacy policy contains a sentence "*we will **not** store your real phone number , name and contacts*", but *PPChecker* identified a path between *<android.provider.ContactsContract$Contacts: android.net.Uri CONTENT_URI>* and *Log.i()*, indicating that the contact information will be stored in the log file. Another app is "hko.MyObservatory_v1_0". Its privacy policy declares "*Users locations would **not** be transmitted out from the app*". However, *PPChecker* found a path from *getLatitude()* to *Log.i()*, meaning that the location information will be stored in log.

False positives. We observed two false positives due to the lack of consideration the context. For example, *PPChecker* found from the code of "com.zoho.mail" that it will access the account information. However, it correlated this behavior with the sentence "*We also do not process the contents of your user account for serving targeted advertisements*" mistakenly, and thus raised an alert of incorrect privacy policy. Actually, there is another sentence in this app's privacy policy saying "*We may need to provide access to your Personal Information and the contents of your user account to our employees*". In other words, this app does access the account information and its privacy policy has correctly declared such behavior.

False negatives. We randomly selected 20 apps to check whether *PPChecker* causes false negatives. The negative sentences in privacy policies, the permissions inferred from descriptions, and the information collected/retained in code were inspected at the same time. We did not find any false negative.

E. Revealing Inconsistent Privacy Policy

We used three performance metrics to evaluate *PPChecker*'s performance in revealing inconsistent privacy policy, including *Precision*, *Recall*, and *F1-Score* [46].

TABLE IV. *PPChecker*'S PERFORMANCE OF DETECTING INCONSISTENT PRIVACY POLICY

Sentence Category	TP	FP	Precision	Recall	F1-Score
$Sents_{collect,use,retain}$	41	5	89.1%	91.7%	90.4%
$Sents_{disclose}$	39	4	90.6%	92.3%	91.4%

Table IV shows the performance of *PPChecker* when detecting the inconsistency between app's privacy policy and third-party lib's privacy policy. The questionable apps caused by $Sent_{disclose}$ are on a separate row because we observed that nearly half of inconsistent statements use $VP_{disclose}$ as their main verbs. Through manual verification, we found that there are in total 75 questionable apps.

For the inconsistencies in $Sents_{collect}$, $Sents_{use}$, and $Sents_{retain}$, *PPChecker* found 46 apps whose privacy policies are inconsistent with their third-party libs' privacy policies. Manual checking shows that 41 apps really contain such inconsistencies, and therefore the precision is 89.1%. For the inconsistencies in $Sents_{disclose}$, *PPChecker* found privacy policies of 43 apps are inconsistent with third party lib privacy policies. We manually inspected those apps and found that only 4 apps are false alerts. Therefore the precision is 90.7%.

False positives. We found that the false positive is due to ESA that may incorrectly regard two different texts as the same thing. For example, the privacy policy of app `com.StaffMark` has the sentence *"do not transmit that information over the internet"*, and the privacy policy of lib Admob(google) contains the sentence "*We will share personal information with companies*". ESA matches the "information" in former to the "personal information" in latter, and regards them as the same thing by mistake.

False negatives. To check false negative, we randomly select 200 app from the data set. We first extract all negative sentences that involve private information from each app privacy policy, and list all positive sentences that involve private information from each third-party lib privacy policy. Then, we manually inspect these sentences. For $Sents_{collect}$, $Sents_{use}$, and $Sents_{retain}$, 12 out of 200 apps have inconsistent privacy policies and *PPChecker* detects 11 apps. Hence, the recall rate is 91.7%. Moreover, for $Sents_{disclose}$, we found that 13 apps

have inconsistent privacy policies and *PPChecker* detects 12 apps. Therefore, the recall rate is 92.3%.

The false negative is due to the incompleteness of our verb set. For instance, the app `com.starlitt.disableddating` declares the sentence "*we will not display any of your personal information*". *PPChecker* fails to match such sentence since "display" is not included in our extracted patterns. We will use the synonyms of major verbs to tackle this issue in future work.

F. Summary of the experimental result

For 1,197 apps in our dataset, *PPChecker* found 282 apps (23.6%) that has at least one problem. More precisely, 222 apps have incomplete privacy policies where 64 apps were detected through descriptions and 180 apps were detected through code. Moreover, *PPChecker* identified 4 apps with incorrect privacy policies where 2 apps were discovered through descriptions and 4 apps were discovered through code. *PPChecker* also revealed 75 apps having inconsistent privacy policies.

VI. Discussion

Being the first step towards assessing the trustworthiness of apps' privacy policies, *PPChecker* has successfully revealed many questionable privacy policies by using the state-of-the-art NLP and static code analysis techniques. The accuracy of *PPChecker* can be further improved from two aspects.

First, employing advanced approaches to understand complex sentences. For example, the constraints in complex sentences, such as "without your consent", "if you do not allow us to", etc., may affect the actual meaning of the sentence. We will create models for these constraints and then adjust the meaning of the corresponding sentence if necessary in future work. Moreover, we will take into account the context information of a privacy policy. Second, due to the limitation of static code analysis, *PPChecker* may miss some source-to-sink paths. For example, if a URI string is involved in a complicated string process, *PPChecker* cannot determine whether it is a sensitive one or not. One potential approach is to conduct dynamic analysis for verifying the result of static analysis.

VII. Related Work

A. Privacy policy analysis

Existing studies usually use a small number of pre-defined patterns to analyze privacy policy. Brodie et al. created a set of grammars and used NLP to identify the rule elements in privacy policy [29]. Costante et al. defined five patterns and employed the information extraction techniques to discover the information to be collected by websites [18]. *Text2Policy* used pre-defined patterns to extract access control policies from natural-language software documents and resource-access information from functional requirements [17]. It is worth noting that *PPChecker* employs the enhanced bootstrapping algorithm to identify new patterns automatically.

Breaux et al. defined a formal language to find conflicts between privacy policies manually [22], [23]. Moreover, Yamada et al. manually looked for conflicts among the privacy policy of a few online social networks [47]. The major difference between this paper and theirs is that *PPChecker automatically* discovers the inconsistencies to avoid time-consuming and error-prone manual inspection. Zimmeck et al. *Privee* combines crowdsourcing and binary classification techniques to examine web privacy policies [48]. Note that although *Privee* can determine whether a privacy policy contains statements related to information collection, it cannot find out which information will be collected.

Slavin et al. propose a semi-automatic method to find the information which is retained in an app's bytecode but missed in its privacy policy (i.e., incomplete privacy policy) [49]. The major differences between *PPChecker* and [49] include: 1) [49] manually selects information collection related sentences from privacy policy while *PPChecker* accomplishes this task automatically. 2) When analyzing an app's bytecode, [49] only considers APIs, but *PPChecker* takes into account both APIs and URIs. Moreover, *PPChecker* uses the reachability analysis to avoid infeasible code whereas [49] does not do it.

B. Android app analysis

A number of static analysis research has been done on Android apps. *FlowDroid* [37] is a precise context, flow, field, object-sensitive and lifecycle-aware static taint analysis system for Android apps. Lu et al. [40] analysed both used APIs and content providers, and conducted the joint flow analysis technique to find more privacy disclosures. *EdgeMiner* [36] conducts static analysis on Android framework to determine implicit control flow transition. *AsDroid* matches the static analysis result with text extracted from UI components to detect stealthy behaviors in Android apps [50]. *Whyper* adopts NLP techniques to process an app's description to find out suspicious permissions [51]. *AutoCog* creates a semantic model for Android permissions and uses this model to locate permissions that can not be matched by descriptions [41]. *CHABADA* [52] utilizes topic model to process descriptions and group apps, and then identify apps that use abnormal APIs in the same group. We proposed and developed *AutoPPG* to automatically generate privacy policies for Android apps [53].

VIII. Conclusion

To determine whether apps' privacy policies are trustworthy or not, we propose and develop *PPChecker*, the *first* system for automatically identifying three kinds of problems in privacy policy after tackling several challenging issues in understanding privacy policy and contrasting the meaning of an app's privacy policy and its behaviors. We have evaluated *PPChecker* with real apps and privacy policies and found that *PPChecker* can effectively detect questionable privacy policies with high precision. Moreover, the experimental results show that 282 of 1,197 popular apps (i.e., 23.6%) contain at least one kind of problem. This research sheds light on the research of improving and regulating apps' privacy policies.

Acknowledgment

We thank the anonymous reviewers for their quality reviews and suggestions. This work is supported in part by the HKPolyU Research Grant (No. G-UA3X), the Hong Kong GRF/ECS (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396), the Hong Kong ITF (No. UIM/285), Shenzhen City Science and Technology R&D

Fund (No. JCYJ20150630115257892), and China Postdoctoral Science Foundation (No. 2015M582663).

REFERENCES

[1] C. Voskoglou, "Sizing the app economy," http://www.developereconomics.com/report/sizing-the-app-economy/, 2013.

[2] FireEye Inc., "Out of pocket: A comprehensive mobile threat assessment of 7 million ios and android apps," http://goo.gl/p6uzdD, 2015.

[3] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Proc. TRUST*, 2012.

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

[5] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. ACM SOUPS*, 2012.

[6] Google, "Developer privacy policy," https://goo.gl/IiuWEH.

[7] "The need for privacy policies in mobile apps c an overview," http://goo.gl/DtAQts, 2013.

[8] M. Brennan, "California ag sends enforcement letter to developers of popular mobile apps," http://goo.gl/2VQeB5, 2012.

[9] "The California Online Privacy Protection Act (CalOPPA)," http://goo.gl/6HtB4N, 2004.

[10] "Directive 95/46/ec of the european parliament and of the council of 24 october 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data," http://goo.gl/Qgwtle.

[11] FTC, "Mobile privacy disclosures: Building trust through transparency," https://goo.gl/h1gVXQ, 2013.

[12] C. Meyer, E. Broeker, A. Pierce, and J. Gatto, "Ftc issues new guidance for mobile app developers that collect location data," http://goo.gl/FxHuj1, 2015.

[13] R. Balebako and L. Cranor, "Improving app privacy: Nudging app developers to protect user privacy," *Security Privacy, IEEE*, vol. 12, no. 4, 2014.

[14] F. Schaub, R. Balebako, A. Durity, and L. Cranor, "A design space for effective privacy notices," in *Proc. SOUPS*, 2015.

[15] R. Balebako, A. Marsh, J. Lin, J. Hong, and L. Cranor, "The privacy and security behaviors of smartphone app developers," in *Proc. USEC*, 2014.

[16] "Ftc path case helps app developers stay on the right, er, path," https://goo.gl/JKgJT4, 2013.

[17] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural language software documents," in *Proc. FSE*, 2012.

[18] E. Costante, J. Hartog, and M. Petkovic, "What websites know about you," in *Proc. DPM*, 2012.

[19] C. Manning and H. Schutze, *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[20] F. Nielson, H. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 2010.

[21] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. ACM CODASPY*, 2012.

[22] T. Breaux, H. Hibshi, and A. Rao, "Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements," *Requirements Engineering*, vol. 19, no. 3, 2014.

[23] T. Breaux and A. Rao, "Formal analysis of privacy requirements specifications for multi-tier applications," in *IEEE RE*, 2013.

[24] L. Yu, X. Luo, C. Qian, and S. Wang, "Revisiting the description-to-behavior fidelity in android applications," in *Proc. IEEE SANER*, 2016.

[25] "Beautiful soup," http://goo.gl/0Lh7Dk.

[26] "Natural language toolkit," http://www.nltk.org/.

[27] D. Cer, M. Marneffe, D. Jurafsky, and C. Manning, "Parsing to stanford dependencies: Trade-offs between speed and accuracy," in *Proc. LREC*, 2010.

[28] Stanford Parser, "Stanford typed dependencies manual," http://nlp.stanford.edu/software/dependencies_manual.pdf.

[29] C. Brodie, C.-M. Karat, and J. Karat, "An empirical study of natural language parsing of privacy policy rules using the sparcle policy workbench," in *Proc. SOUPS*, 2006.

[30] J. Slankas, X. Xiao, L. Williams, and T. Xie, "Relation extraction for inferring access control rules from natural language artifacts," in *Proc. ACSAC*, 2014.

[31] J. R. Curran, T. Murphy, and B. Scholz, "Minimising semantic drift with mutual exclusion bootstrapping," in *Proc. ACL*, 2007.

[32] "Negative vocabulary word list," http://goo.gl/qX7UtK.

[33] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: toward discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, 2015.

[34] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed android applications," in *Proc. ESORICS*, 2015.

[35] L. Li, A. Bartel, T. Bissyande, J. Klein, Y. Traon, S. Arzt, R. Siegfried, E. Bodden, D. Octeau, and P. Mcdaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proc. ICSE*, 2015.

[36] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," in *Proc. NDSS*, 2015.

[37] 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 *Proc. PLDI*, 2014.

[38] K. Au, Y. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. CCS*, 2012.

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

[40] K. Lu, Z. Li, V. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting," in *Proc. NDSS*, 2015.

[41] 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. ACM CCS*, 2014.

[42] E. Gabrilovich and S. Markovitch, "Computing semantic relatedness using wikipedia-based explicit semantic analysis." in *Proc. IJCAI*, 2007.

[43] "Top 90 popular ad libraries," http://goo.gl/GBhXOi, 2015.

[44] "Top 18 popular social libraries," http://www.appbrain.com/stats/libraries/social, 2015.

[45] "The most popular develop tools," http://www.appbrain.com/stats/libraries/dev, 2015.

[46] M. Bramer, *Principles of Data Mining*, 2nd ed. Springer, 2013.

[47] A. Yamada, T. H.-J. Kim, and A. Perrig, "Exploiting privacy policy conflicts in online social networks," *CMU-CyLab-12-005*, 2012.

[48] S. Zimmeck and S. M. Bellovin, "Privee: An architecture for automatically analyzing web privacy policies," in *Proc. USENIX Security*, 2014.

[49] R. Slavin, X. Wang, M. B. Hosseini, W. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu, "Toward a framework for detecting privacy policy violation in android application code," in *Proc. ICSE*, 2016.

[50] 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. ICSE*, 2014.

[51] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *Proc. USENIX Security*, 2013.

[52] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. ICSE*, 2014.

[53] L. Yu, T. Zhang, X. Luo, and L. Xue, "Autoppg: Towards automatic generation of privacy policy for android applications," in *Proc. ACM SPSM*, 2015.