PPChecker: Towards Accessing the Trustworthiness of Android Apps' Privacy Policies

Le Yu, Xiapu Luo, Jiachi Chen, Hao Zhou, Tao Zhang, Henry Chang, and Hareton K. N. Leung

***Abstract*—Recent years have witnessed a sharp increase of malicious apps that steal users' personal information. To address users' concerns about privacy risks and to comply with data protection laws, more and more apps are supplied with privacy policies written in natural language to help users understand an app's privacy practices. However, little is known whether these privacy policies are trustworthy or not. Questionable privacy policies may be prepared by careless app developers or someone with malicious intention. In this paper, we carry out a systematic study on privacy policy by proposing a novel approach to automatically identify five kinds of problems in privacy policy. After tackling several challenging issues, we implement the approach in a system, named *PPChecker*, and evaluate it with real apps and their privacy policies. The experimental results show that *PPChecker* can effectively identify questionable privacy policies with high precision. Applying *PPChecker* to 2,500 popular apps, we find that 1,850 apps (i.e., 74.0%) 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 thanks to the great driving force from apps. Actually, the global app economy reached $77 billion in 2017 [1] and expected to rise to $110 billion in 2018 [2]. Since the number of various malicious apps (e.g., malware, ransomware, adware, etc.) is also rapidly increasing [3], users are very concerned about the privacy risks when using apps [4], [5]. Although Android lists the permissions required by each app before installation, it is usually difficult for normal users to understand the potential threats by just reading the permissions [6].

App developers can upload a privacy policy to Google Play for declaring what information from users will be collected, used, retained, or disclosed [7]. A survey showed that 76% free apps in Google Play have provided privacy policies in 2012 [8]. Many jurisdictions have enacted the privacy laws to require developers to include privacy policies with their apps, such as, California [9] and its California Online Privacy Protection Act(CalOPPA) [10], the Data Protection Directive(95/46/EC) [11] in European Union, etc. Federal Trade Commission (FTC) suggests mobile developers to prepare privacy policies for their apps [12] and provides guidance [13]. Moreover, Google Play recently required developers to provide a valid privacy policy when the app requests or handles sensitive user or device information. If developers do not meet these policy requirements, Google can "limit the visibility of your app" or even remove the app [14].

Le Yu, Xiapu Luo, Jiachi Chen, Hao Zhou, Tao Zhang, and Hareton K. N. Leung are with the Department of Computing in the Hong Kong Polytechnic University. Tao Zhang is also with the College of Computer Science and Technology in Harbin Engineering University. Henry Chang is with the Department of Law in the University of Hong Kong.

Unfortunately, it is not easy to prepare an accurate privacy policy for an app due to many reasons [15], [16]. For instance, it is not uncommon that the author of a privacy policy is different from the app developer if 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 mention using them. However, Balebako et al. find that around half of developers do not know for sure what information will be collected by third-party libs in their apps [17], because few third-party libs provide source code and they are often less transparent about their data collection practice. 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 [18].

Therefore, given an app and its privacy policy, user may wonder whether they can trust the privacy policy. Although manually dissecting apps and scrutinizing the privacy policies could answer this question, it is time-consuming. In this paper, we propose a novel approach and develop a system named *PPChecker* to automatically identify problems in privacy policy. The output of *PPChecker* can facilitate app stakeholders to spot issues in their privacy policies, help normal users to determine the trustworthiness of apps, and assist app owners and regulators like FTC to identify questionable apps. The checking of inaccurate privacy policies can also be used to detect malicious apps. For example, the hidden behaviors detected in an incomplete privacy policy may come from a malicious component of repackaged app [19]. Moreover, an adversary can create an incorrect privacy policy to cheat users.

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 the meaning of a privacy policy or extract useful information from it [20], [21]. Moreover, both the app's and the third-party libs' privacy policies should be analyzed in order to spot any 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 then contrast the behaviors with the information extracted from the privacy policy. To tackle these challenging issues, *PPChecker* employs natural-language processing (NLP) techniques [22] to analyze privacy policies, and adopts program analysis approaches [23] to inspect apps (Section III). Moreover, we define five kinds of common problems in privacy policies (Section II-B) and propose new algorithms to detect them (Section IV).

In summary, our major contributions include:

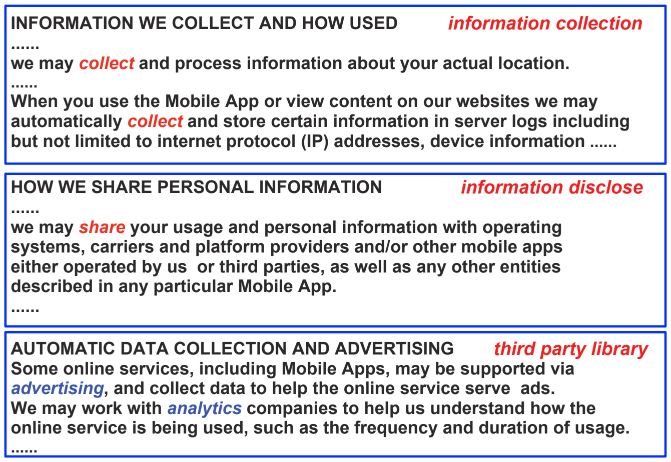
- We define five kinds of common problems in privacy policies and propose new algorithms to identify them.
- We design and develop *PPChecker*, a novel system that adopts NLP and program analysis techniques to automatically identify the problems in privacy policy.
- We conduct careful evaluation on *PPChecker* by using real apps along with their privacy policies. The experimental results 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 the 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, Description, and What's New

Privacy policy informs users what, when, why, and how information will be collected. For example, Fig. 1 shows a portion of an app's (i.e., Golf Live Extra) privacy policy. It first says "*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 personal information to be collected, used, retained or disclosed by an app as described in its privacy policy.


INFORMATION WE COLLECT AND HOW USED *information collection*
......
we may *collect* and process information about your actual location.
......
When you use the Mobile App or view content on our websites we may automatically *collect* and store certain information in server logs including but not limited to internet protocol (IP) addresses, device information ......

HOW WE SHARE PERSONAL INFORMATION *information disclose*
......
we may *share* your usage and personal information with operating systems, carriers and platform providers and/or other mobile apps either operated by us or third parties, as well as any other entities described in any particular Mobile App.
......

AUTOMATIC DATA COLLECTION AND ADVERTISING *third party library*
Some online services, including Mobile Apps, may be supported via *advertising*, and collect data to help the online service serve ads.
We may work with *analytics* companies to help us understand how the online service is being used, such as the frequency and duration of usage.
......

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

The description of an app usually introduces the functionalities, features and benefits of the app [24], and What's new (denoted as WsN) informs users about the changes (e.g., new features, fixed bugs, etc.) [25].

B. Problem Definition

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

(1) Incomplete privacy policy. Such privacy policy does not declare all privacy-related behaviors. For example, the app's (`com.dooing.dooing`) description has a sentence "*Location aware tasks will help you to utilize your field force in optimum way.*" indicating the use of location information. Its class `com.dooing.dooing.ee` calls location-related APIs including *getLatitude()* and *getLongitude()*. However, its privacy policy does not mention that it will collect location information or personal information.

(2) Incorrect privacy policy. The privacy policy is incorrect if it declares that the app does *not* collect, use, retain, or disclose personal information, but the app does. For instance, the app's (`com.easyxapp.secret`) privacy policy declares "*we will **not** store your real phone number , name and contacts*". However, we find 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) Imprecise privacy policy. The imprecise privacy policy does not clearly describe the privacy related behaviors. In particular, we focus on three kinds of imprecise privacy policy. The first kind of imprecise privacy policy does not clearly specify the personal information accessed by the app. Instead, they use general terms to represent such information. For example, the app (`com.ivc.starprint`) collects device ID in code. However, its privacy policy only mentions that "*we may also access other **personal information on your device**, such as your phone book, calendar, in order to provide services to you*". In other words, it does not clearly point out that the app will collect device ID.

The second kind of imprecise privacy policy lists the personal information to be accessed, but the app only accesses part of them. User may wonder why the app needs to collect so much personal information when reading such privacy policies. For instance, the privacy policy of the app (`com.liquifyble.main`) says that device ID will be collected (i.e., "*'When you access the Service by or through a mobile device, we may collect certain information automatically, including, but not limited to, the type of mobile device you use, your mobile devices unique device ID*"). However, the app neither requests `READ_PHONE_STATE` permission nor gets device ID in its code.

The third kind of imprecise privacy policy uses adverb clause of condition to describe the condition under which the privacy-related behavior will be conducted, but the app does not check whether the condition is satisfied. For example, the privacy policy of the app (*mobisocial.omlet*) declares that "*If you click go, we will collect information about your geographic location*". Note that "go" refers to a button in this app. However, after checking the code, we found that the location-related API (*LocationManager.getLastKnownLocation()*) is called by the *com.baidu.android.pushservice.b.e.h()* method, instead of the button's callback function(i.e., *onClick()*).

(4) Inconsistent privacy policy. The app's privacy policy should be consistent with the privacy policies from the other two sources. First, if an app integrates 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 do so, the app's privacy policy is inconsistent. For example, the privacy policy of the app (`com.imangi.templerun2`) claims that it does *not* use or collect location information. However, one embedded third-party lib (`Unity3d`) says that it will receive location information, thus causing the inconsistency. Second, the privacy policy (provided in Google Play) should be consistent with the in-app privacy policy, which is shown when the app is being used. If the information collected by these two privacy policies are different, the app has inconsistent privacy policy. For example, for the app (`com.jimbl.suitcaseluggagelistfrgoog`), its privacy policy in the market (*http://www.jimbl.com/privacy1*) and in-app privacy policy (*http://www.jimbl.com/privacy2*) are different. The formal declares that "*The Application will collect the following information ... unique device ID ...*" whereas the latter does not mention such behavior at all.

(5) User-unfriendly privacy policy. We focus on two issues in such privacy policy. First, the language of privacy policy is different from the language used for the description. We assume that the user who installs an app understands the app's description and not all users understand two languages. A recent report from Hong Kong's Office of the Privacy Commissioner for Personal Data pointed out such issue after manual inspection [26]. For example, the description of the app `chat.ola.vn` is in English whereas its privacy policy is stated in Vietnamese. Second, some privacy policies' readability is low so that users cannot understand it.

III. PPChecker

We give an overview of *PPChecker* in Section III-A and detail its major modules in Section III-B, III-C, and IV.

A. System Overview

As shown in Fig. 2, *PPChecker* takes in an app's description, what's new information (WsN), third-party libs' privacy policies, app's privacy policy, and apk file. 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 imprecise or not. If so, it lists the imprecise sentences; 4) whether the privacy policy is inconsistent or not. If so, it lists the inconsistent sentences and the relevant third-party lib's privacy policy; 5) whether the privacy policy is user unfriendly or not. If so, it lists the language of the privacy policy and the readability rating of the privacy policy. *PPChecker* consists of three major modules, including:

(1) Privacy policy analysis module (Section III-B). It analyzes a privacy policy to determine what information will or will 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, retain, or disclose personal information. Note that this module cannot differentiate the used information from the collected/retained/disclosed information because it is difficult to determine whether the information is used or not after it has been collected from the device.

(3) Description and what's new analysis module (Section III-D). It analyzes the description and the WsN to identify the permissions/personal information that the app will use.

(4) Problem identification module (Section IV). Based on the result of **(1)** and **(2)**, this module identifies incomplete privacy policy (IV-A), incorrect privacy policy (IV-B), imprecise privacy policy (IV-C). By comparing the app's privacy policy with its in-app privacy policy and its third-party libs' privacy policies, it detects inconsistent privacy policy (IV-D). By analyzing the language and readability of app privacy policy, it identifies user-unfriendly privacy policy (IV-E).

B. Privacy Policy Analysis Module

1. Main Verbs, Sentences and Resources

TABLE I
MAJOR SYMBOLS RELATED TO PRIVACY POLICY. NOTE: $* \in \{collect, use, retain, disclose\}$

Symbol	Meaning
$VP_{collect}$	the verbs describe that one party access, collect, or acquires data from another party.
VP_{use}	the verbs describe that one party uses data from another party for some function.
VP_{retain}	the verbs describe that one party keeps the data collected from another party.
$VP_{disclose}$	the verbs describe that one party transfers the collected data to another party.
$AppSent_*$	a sentence in app's privacy policy whose main verb $\in VP_*$
Res_*^{AppPP}	resources that the app will * according to an app's privacy policy
$\overline{Res_*^{AppPP}}$	resources that app will **not** * according to an app's privacy policy
$LibSent_*$	a sentence in lib's privacy policy whose main verb $\in VP_*$
Res_*^{LibPP}	resources to the lib will * according to the lib's privacy policy

Table I lists the symbols related to privacy policy. We summarize four types of main verbs, which are commonly used in privacy policies, as suggested by [27], [28], including:

• **Collect verbs**. They describe that one party accesses, collects, obtains data from another party, such as "collect", "gather", etc. We use $VP_{collect}$ to indicate such verbs.

• **Use verbs**. They depict that one party uses data from another party for a specified purpose, such as "use", "process", etc. We use VP_{use} to 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. We use VP_{retain} to represent such verbs.

• **Disclose verbs**. They indicate that one party transfers, moves, or sends data to another party, such as "disclose", "share", etc. We use $VP_{disclose}$ to stand for such verbs.

These four kinds of verbs have different semantic meaning and are essential to tracing the data flows described in privacy

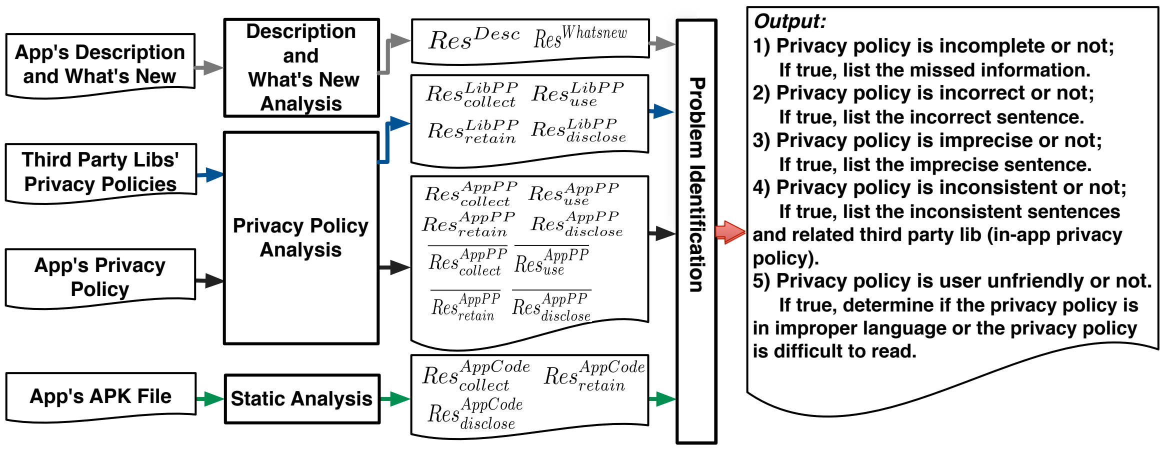

App's Description and What's New
Third Party Libs' Privacy Policies
App's Privacy Policy
App's APK File
Description and What's New Analysis
Privacy Policy Analysis
Static Analysis
Res^{Desc} $Res^{Whatsnew}$
$Res^{LibPP}_{collect}$ Res^{LibPP}_{use} Res^{LibPP}_{retain} $Res^{LibPP}_{disclose}$
$Res^{AppPP}_{collect}$ Res^{AppPP}_{use} Res^{AppPP}_{retain} $Res^{AppPP}_{disclose}$
$\overline{Res^{AppPP}_{collect}}$ $\overline{Res^{AppPP}_{use}}$ $\overline{Res^{AppPP}_{retain}}$ $\overline{Res^{AppPP}_{disclose}}$
$Res^{AppCode}_{collect}$ $Res^{AppCode}_{retain}$ $Res^{AppCode}_{disclose}$
Problem Identification
Output:
1) Privacy policy is incomplete or not; If true, list the missed information.
2) Privacy policy is incorrect or not; If true, list the incorrect sentence.
3) Privacy policy is imprecise or not; If true, list the imprecise sentence.
4) Privacy policy is inconsistent or not; If true, list the inconsistent sentences and related third party lib (in-app privacy policy).
5) Privacy policy is user unfriendly or not. If true, determine if the privacy policy is in improper language or the privacy policy is difficult to read.

Fig. 2. System overview of *PPChecker*

policy [27]. To construct the verb set, we download the keywords listed in [28] and put them in different verb lists. We also use WordNet to find the synonyms of these verbs and add them to the verb lists. We release the total list of such verbs in [29] so that other researchers can use them.

Based on the four types of verbs, we define other symbols. $AppSent_*$ 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, we will use $LibSent_*$ and $LibSents_*$ to represent those sentences in a third-party lib's privacy policy.

By analyzing the sentences, *PPChecker* identifies the personal information handled by main verbs. For positive sentences, we use Res_*^{AppPP} to denote the set of personal information that will be collected, used, retained, and disclosed by app according to app's privacy policy. Similarly, we use Res_*^{LibPP} to denote the set of personal information that will be collected, used, retained, and disclosed by third-party libs according to libs' privacy policies.

A privacy policy may use negative sentences. For example, "*we will not collect*" presents the opposite meaning of "*we will collect*". The former is negative sentence. We utilize $\overline{Res_*^{AppPP}}$ to denote the set of personal information that will *not* be collected, used, retained, or disclosed, according to an app's privacy policy. We do not consider the negative sentences in third-party lib's privacy policy, because if the app's privacy policy declares collecting personal information, it will not be inconsistent no matter whether the lib collects this information or not.

2. Steps in Privacy Policy Analysis

Fig. 3 shows the procedure of inspecting a privacy policy, which involves seven steps detailed as follows.

Step 1: Sentence extraction. *PPChecker* extracts the content from each privacy policy in HTML, and removes non-ASCII symbols and some meaningless ASCII symbols using Beautiful Soup [30]. We currently just examine privacy policies in English. Then, we use the natural language toolkit (NLTK) [31] to divide the text into sentences. Since NLTK splits an enumeration list into individual sentences, it may cause errors. For example, the sentence "*... collect the following information: your name; your IP address; your device ID.*" is 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 ":"or ";", or the current sentence starts with lowercase letters, *PPChecker* appends the current sentence to the previous one. Finally, *PPChecker* turns all letters into lower case.

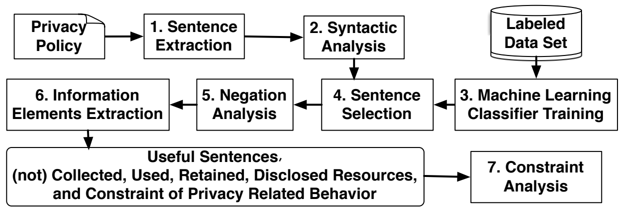
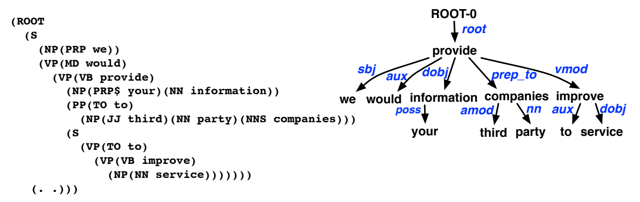

Privacy Policy
1. Sentence Extraction
2. Syntactic Analysis
Labeled Data Set
3. Machine Learning Classifier Training
4. Sentence Selection
5. Negation Analysis
6. Information Elements Extraction
Useful Sentences, (not) Collected, Used, Retained, Disclosed Resources, and Constraint of Privacy Related Behavior
7. Constraint Analysis

Fig. 3. 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 [32] to obtain its parse tree and dependency relations. For example, Fig. 4 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 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. 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, $auxpass$ means passive auxiliary [33]. For example, from the sentence "*your information will be collected.*", We can find $nsubjpass$ dependency relation between "information" and "collected" and discover $auxpass$ dependency relation between "be" and "collected".

The syntactic information is used in the following classifier training and the information elements extraction step.



```
(ROOT
  (S
    (NP(PRP we))
    (VP(MD would)
      (VP(VB provide)
        (NP(PRP$ your)(NN information))
        (PP(TO to)
          (NP(JJ third)(NN party)(NNS companies)))
        (S
          (VP(TO to)
            (VP(VB improve)
              (NP(NN service)))))))
   (. .)))
```

Fig. 4. Syntactic information of sentence: "*we would provide your information to third party companies to improve service*".

Step 3: Classifier training. In the conference version [34], we use a series of seed patterns to find the semantic patterns from privacy policy corpus. These patterns are used to identify information collection, usage, retention and disclosure related sentences. However, if the pre-defined seed patterns are incomplete or incorrect, the discovered patterns cannot identify the sentences related to information collection, usage, retention, and disclosure with a high recall rate and high precision. To overcome this limitation, in this paper, we propose using machine learning classifier to identify these sentences automatically.

More precisely, we let the feature set include unigrams, bigrams, trigrams, and type dependency related between words. The TF-IDF (term frequency-inverse document frequency) [35] value is calculated to measure the weight of each word. Such value is commonly used in information retrieval to measure how important a word is. TF (i.e., term frequency) refers to how frequently a term occurs in a document. IDF (i.e., inverse document frequency) means the terms that appear many times have little importance (e.g., "an", "of"). We select four classifiers, including Max Entropy [36], SVM [37], naive bayes [38], and Random Forest [39], and select the best one according to the 10-fold cross validation result on our data set. The result is described in Section V-B. For the sentences identified by the classifier, we only keep those sentences that contain collect, use, retain, and disclose verbs (Section III-B.1).

Step 4: Sentence selection. *PPChecker* utilizes the trained classifier to identify sentences from privacy policies. The matched sentences are regarded as *useful sentences*, and others will be discarded.

Step 5: Negation analysis. *PPChecker* determines whether a sentence is negative by checking the existence of negation words in two places [20]. 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 [40], 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: Information elements extraction. From each useful sentence, we look for four elements, including main verb, action executor, resource, and constraint. For example, in the sentence: *"we will provide your information to third party companies to improve service if you ..."*, the main verb is "provide", the subject is "we", its object is "your information", and its constraint is "if you ...". The subject, main verb, resource, and constraint are used in the problem identification module (Section IV-A).

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 (e.g., "provide" in Fig. 4). 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 the main verb (e.g., "we" in Fig. 4). The resource is the data used by the action executor (e.g., "information" in Fig. 4). 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 the main verb. For instance, in the sentence "*your location will be collected*", the resource "your location" is the subject. For each extracted resource, we check the category of the corresponding main verb (by using the pre-defined verb set described in Section III-B.1) to determine the resource belongs to collected information ($Res^{AppPP}_{collect}$), used information (Res^{AppPP}_{use}), retained information (Res^{AppPP}_{retain}), or disclosed information ($Res^{AppPP}_{disclose}$). Specially, if the negation analysis result shows the sentence is a negative sentence, we put the extracted resource to the information that privacy policy declares not to collect ($\overline{Res^{AppPP}_{collect}}$), use ($\overline{Res^{AppPP}_{use}}$), retain ($\overline{Res^{AppPP}_{retain}}$), or disclose ($\overline{Res^{AppPP}_{disclose}}$).

The constraint refers to the condition under which the privacy-related behavior will be conducted. In this paper, we only consider the constraint described in the adverb clause of condition (e.g., clauses starting with "*if, unless, in case, only if*") [41], and leave the investigation of other kinds of adverb clauses in future work.

When identifying the personal information mentioned in privacy policies, we find that some sentences contain *refinement* phrases [42]. The *refinement* phrase refines one general concept with more specific concepts [27], [42]. The specific concept refers to a specific kind of information accessed by the app (e.g., device ID) and the general concept is an abstraction of the specific concept (e.g., device information). For example, the sentence "*Personal information may include your name, user identification number, or email address.*" refines "personal information" with a few specific concepts: "name", "user identification number", "email address". We propose the following method to process them.

▷ We employ the syntactic patterns proposed by Girju et al. [43] (listed in Tab. II) to identify the part-whole relations described by *refinement* phrases. For instance, the sentence "*Personal information may include your name, user identification number, or email address*" matches the first pattern of Tab. II. Then, we can find that there is a part-whole relation between the subject and the objects of the verb "include". Therefore, the general concept is "personal information", and the specific concepts include "name", "user identification number, and "email address".

▷ We extract all sentences from the privacy policy and store them in a list and then all sentences that contain *refinement* phrase. If the i sentence in the sentence list contains *refinement* phrase, we extract the $(i-1)$ and $(i+1)$ sentences from the sentence list of privacy policy. We use the trained classifier (Section III-B Step 3) to determine if these two sentences are related to information collection, usage, retention, or disclosure or not. If the $(i-1)$ or $(i+1)$ sentence is regarded as information collection, usage, retention, or disclosure and the general concept of the *refinement* phrase is also mentioned in it, we conclude that the corresponding specific concepts will be collected, used, retained, or disclosed.

We use the following two sentences as an example to illustrate this procedure: "*We collect your personal information. The personal information includes your name, your address, and device information*". After finding the second sentence that contains *refinement* phrase, we apply the trained classifier to the first sentence and find that it is related to information disclosure. Since the first sentence declares that the general concept (i.e., "personal information") will be collected, we infer that the specific concepts (i.e., "name", "address", "device information") will also be collected.

Step 7: Constraint analysis. After identifying the constraints by extracting the adverb clauses of condition in useful sentences, we first filter out the constraints that are not relevant to the app's behaviors. More precisely, we ignore the constraints that cannot be achieved by the app, including contacting developer, asking developer question, using/cancelling service, visiting website, legal requirement, determining whether the user is a child. To filter out such constraints, we built up a blacklist that contains a series of (verb, noun) pairs (e.g., ("contact", "us"), ("change", "policy")) after reading 300 randomly selected adverb clauses of condition. Then, for each useful sentence (identified in Step 4: Sentence selection), if it contains constraints, we extract all the (verb, noun) pairs based on the typed dependency relations. For example, for the constraint "*if you enable location service*", two (verb, noun) pairs (i.e., ("enable", "you") and ("enable", "location service")) will be extracted since there are *nsbj* and *dobj* dependency between words. After that, we use ESA [44] to calculate the semantic similarity between each (verb, noun) pair extracted from the constraint and each (verb, noun) pair in the blacklist. If the similarity is higher than a threshold (by default, 0.67 as used in [45]), we ignore the constraint. Otherwise, we analyze its content. To get the semantic similarity of two texts, ESA maps each of them to a vector representation using a knowledge base, and then calculates the similarity of the two vectors.

For the apps in the data set, we discovered 173 useful sentences that contain constraints. We summarize the topics of these constraints to find out the types of constraints that can be checked in code. In detail, we extract the topic words from these constraints by using Mallet [46] to identify 10 topic words from the document. Mallet is a topic modeling program that can select words from baskets of words (each basket related to a topic) [47]. By reading these topic words and their related constraints, we determine two types of constraints that can be checked in code (Tab. III).

The first type of constraint is relevant to enabling service and has the template like "If users enable feature/-function/service, the app will access the information". In the app, the developer will check whether the corresponding feature/function/service has been enabled before accessing sensitive information. These features/functions/services can be divided into two categories. One category includes the services provided by Android system (e.g., location, bluetooth). Another category includes the feature/function implemented by the developer. For example, the privacy policy of the app (`com.media1908.lightningbug`) contains the sentence "*If you do enable the local weather feature, your approximate location is collected on a periodic basis*". The "local weather feature" is implemented by the developer.

The second type of constraint is relevant to UI callback and has the template like "If users click/press UI element, the app will access the information". The UI element refers to the View classes included in the layout (e.g., *Button*, *TextView*) [48]. After a user presses the UI element, such apps may directly access sensitive information in the callback of the UI element or launch other activities to access the sensitive information.

C. Static Analysis Module

Table IV includes five kinds of information extracted from app code. Given an app, *PPChecker* conducts static code analysis on its *dex* file to determine the following information: (1) personal information collected by the app (i.e., $Res^{AppCode}_{collect}$) and third-party lib (i.e., $Res^{LibCode}_{collect}$); (2) personal information retained by the app (i.e., $Res^{AppCode}_{retain}$) and third-party lib ($Res^{LibCode}_{retain}$). (3) if the information is collected or retained by third party lib, we regard it as disclosed information (i.e., $Res^{AppCode}_{disclose} = Res^{LibCode}_{collect} \cup Res^{LibCode}_{retain}$). To check the constraints of privacy related behavior, this module will extract the statements that will affect the invocation of sensitive APIs/URIs. To get the in-app privacy policy, this module also extracts the URL of privacy policy from code.

TABLE IV
MAJOR SYMBOLS RELATED TO CODE.

Symbol	Meaning
$Res^{AppCode}_{collect}$	resources app will collect according to code
$Res^{AppCode}_{retain}$	resources app will retain according to code
$Res^{LibCode}_{collect}$	resources lib will collect according to code
$Res^{LibCode}_{retain}$	resources lib will retain according to code
$Res^{AppCode}_{disclose}$	resources app will disclose according to code

1. Conducting the Static Bytecode Analysis

We develop the static analysis module based on our static analysis tool, *VulHunter* [49], and improve 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 employ our unpacking tool *PackerGrind* [50], [51] to recover the `dex` file. If Java reflection is used to invoke methods, we utilize DoridRA [52] to recognize the invoked methods. By parsing the `AndroidManifest.xml` file and the `dex` file, *PPChecker* constructs an Android property graph (APG) [49] that integrates abstract syntax tree (AST), interprocedure control-flow graph (ICFG), method call graph (MCG), and data dependency graph (DDG) of the app, and stores them into a graph database. Then, *PPChecker* can determine the collected and retained information by performing queries on APGs.

To enhance the accuracy of static analysis, we employ *IccTA* [53] to identify the source and the target of an intent,

TABLE II
SYNTACTIC PATTERNS OF REFINEMENT PHRASES

#	Pattern	Example
1	`obj` *contain|include* `parts`.	"*Personal information includes name and address*"
2	`obj` *consist of|be made of* `parts`.	"*Personal information is made of name and address*"
3	......`obj`......, including `parts`.	"*Personal information will be collected, including name and address*"
4	`obj` *such as* `parts`	"*We will collect personal information such as name and address*"

TABLE III
CONTENT TYPE OF CONSTRAINT

#	Content Type	Example
1	If users enable feature/function/service, the app will access the information.	"*If you choose to enable Allow Others to Add feature, your telephone number is used*"
2	If users click/press UI element, the app will access the information.	"*In some of our games you may find a camera button, once pressed it will simply take a picture of what you are seeing.*"

and utilize *EdgeMiner* [54] to determine the implicit callbacks (e.g., from *setOnClickListener()* to *onClick()*). The data dependency between statements $s1$ and $s2$ means that a variable v is defined in $s1$ and used in $s2$. Moreover, for the source-sink paths identified by *FlowDroid* [55], we also add data dependency relation between the source statement and sink statement.The sources refer to the sensitive APIs/URIs, which will be described in the next sub-section. The sinks are 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()* to get device ID. The other one is through content providers [56], such as calling *android.content.ContentResolver.query()* with *content://com.android.calendar* to access calendar information. We will describe how *PPChecker* handles them in the following paragraphs, respectively. By referring PScout [56] and checking Android development documents, we select 152 sensitive APIs covering the information about device ID (9), IP address (5), cookie (10), location (31), account (30), contact (35), calendar (6), telephone number (2), IMEI (5), camera (4), audio (6), sim card number (1), call log (1) and app list (7). Such information is commonly listed in the privacy policy. We select 12 URI strings along with 615 URI fields from the data set in [56].

The sensitivity of content provider related operations is determined by the parameters. For example, by querying the content provider with URI *content://call_log* or *content://com.android.calendar*, we can collect the call log or the calendar from the device respectively. To find out the information accessed by using URIs, after locating the statements calling the query function, *PPChecker* determines the corresponding URIs by applying the inter-procedure constant propagation (i.e., constant folding) algorithm proposed by Lu et al. [57]. Constant propagation is a compiler technique that simplifies the constant expressions [58]. Here, we apply this technique to identify all possible constant values that can be used as the parameters of content provider related operations. The detailed analysis includes the following steps:

Step 1: Similar to the algorithm proposed by Lu et al. [57], we leverage the data dependency graph to identify all the statements used to construct the URI. Other statements will not be analyzed. In detail, we traverse along the data dependency graph from the statement querying the content provider. The traversal stops at the statement that only uses constant values. All the statements appeared on the path and the data dependency relations between them are recorded. We stored them as a directed graph.

Step 2: To infer the possible values of URI, we perform *post-order* DFS traversal on this directed graph. The traversal starts from the statement that performs content provider related operation. For a statement s, the *post-order* traversal is to ensure that the possible values of the variable used in it has been analyzed. For each statement, since we have got possible values of the variables used in it, we can infer the possible values of the variables defined in it.

Step 3: We collect a set of possible constants that can be used to construct URI. We look for the sensitive URIs in them and record the sensitive URIs used in code. An example is the code snippet shown in Fig. 5: The URI parameter used in line 9 is constructed by using line 2, 4, 7. The possible values for $v5$ include $\{$ "*content://com.android.calendar*", "*content://call_log*"$\}$.

```
1  ...
2  String v5 = "content://";
   ...
3  if(...){
4     v5 = v5 + "com.android.calendar";
5  }
6  else{
7     v5 = v5 + "call_log";
8  }
9  $activity.getContentResolver().query(Uri.parse(v5),
                                    v4, null, null, null);
```

Fig. 5. Example of traditional constant folding

Since some sensitive APIs may not be called by the app (e.g., infeasible code), we perform the reachability analysis [59] by using inter-procedure control flow graph (ICFG) to filter them out. In detail, we start traversal from apps' entry points, including life-cycle callbacks (e.g., *Activity.onCreate()*)

and UI related callbacks (e.g., *onClick()*). If we can find a path from the entry points to the statements that invoke sensitive APIs, we regard these statements as feasible.

To determine the type of personal information collected by the app, we map the sensitive APIs and the URI strings to personal 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 mapping between URI fields and permissions [56], we map these fields to the personal 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

By using the data dependency graph, we perform static taint analysis on sensitive APIs/URIs to determine the retained information. More precisely, after finding the statements that use sensitive APIs/URIs, we traverse the data dependency graph (DDG) by using data dependency relation to find the path from source (i.e., the statements that use sensitive APIs/URIs) to sink (i.e., log, network, file, SMS, bluetooth related APIs). If we can find the path from source to sink, we think the information will be retained by the app. Since *FlowDroid* [55] is a state-of-the-art static analysis tool for apps, we further integrate the source-to-sink paths found by *FlowDroid* when creating the data dependency graph. The major difference between our system and *FlowDroid* is that:
(1) *FlowDroid* does not consider the data flow caused by the intent sent by components. The static taint analysis of *FlowDroid* stops at the statement that sends out intent (e.g., *startActivity()*). To overcome this limitation, when it comes to a statement that sends out intent, *PPChecker* leverages the result of *IccTA* [53] to identify the target component of the intent. After identifying the life-cycle method of the target component (e.g., *Activity.onCreate()*), *PPChecker* searches the statement that retrieves the intent (e.g., *Activity.getIntent()*) and continues the traversal from this statement.
(2) *FlowDroid* cannot perform constant propagation to find out the URIs used in code. After finding the content provider related operations (e.g., *ContentResolver.query()*), *PPChecker* first conducts constant propagation to identify the URIs used in code and then performs static taint analysis to determine if the information is retained or not.

For instance, Fig.6 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). This path indicates that the information of installed apps will be retained in log.

4. Recognizing the User of the Information

After finding the sensitive API/URI used in code, we first locate the corresponding method. Then, we conduct backward traversal on the method call graph to find all other methods that will call this method. The traversal starts from this method and stops at the entry points of Android (i.e., life-cycle methods of Android components and callbacks of UI widgets). We leverage the class names of these methods to determine whether the API/URI is used by the app or third-party libs. If the privacy related behavior is triggered by callback of the UI widget (e.g., *onClick()*), the user of this behavior is the Activity that contains the widget. Since developers can register UI callbacks dynamically (i.e., in code) or statically (i.e., in layout xml file), we cannot directly identify the Activity associated with the callback. We process these two cases separately to discover the corresponding Activity.

▷ If the developer dynamically registers the callback in code, we first search the listener object (e.g., *View.OnClickListener*) that implements this callback in code. Then, we leverage the DDG to find the widget object that uses the listener (e.g., by calling *setOnClickListener()*). Finally, the method that contains widget object and the corresponding activity are the user of the corresponding information.

▷ If the developer statically registers the callback in the layout xml file, we first obtain the resource ID of the layout. Then, we search the AST to locate the Activity that uses the layout resource ID to set the activity content view. We regard this Activity as the user of the API/URI.

Li et al. [60] harvest commonly used libraries from 1.5 million apps and get a class name prefix list containing 240 class names. If the developer uses obfuscation techniques to hide the class names of third-party libs, we employ LibScout [61] to detect the integrated third-party libs. LibScout extracts a variant of Merkle tree from known lib code to construct the lib profiles. Then, it matches the lib profiles with the app profile to find out the integrated third-party libs. For each sensitive API/URI, if the class name prefix of the corresponding class is included in the class name prefix list, we regard that the information is collected or retained by a third-party lib ($Res^{LibCode}_{collect}$ and $Res^{LibCode}_{retain}$). Otherwise, the information is collected/retained by the app ($Res^{AppCode}_{collect}$ and $Res^{AppCode}_{retain}$). We regard the information collected or retained by third-party lib as the information disclosed to third-party lib (i.e., $Res^{AppCode}_{disclose} = Res^{LibCode}_{collect} \cup Res^{LibCode}_{retain}$).

5. Extracting the Constraints

To check the constrains of privacy related behaviors, we use the program slicing technique [62] to extract the statements that will affect the invocation of sensitive APIs/URIs.

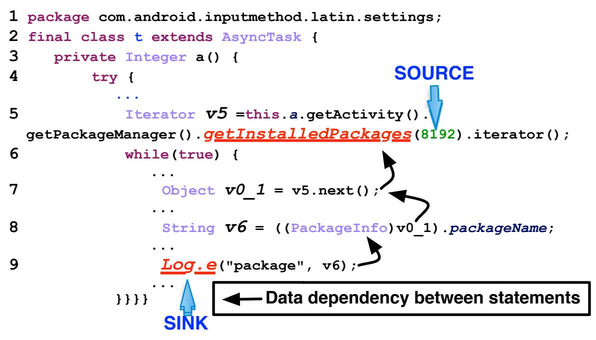
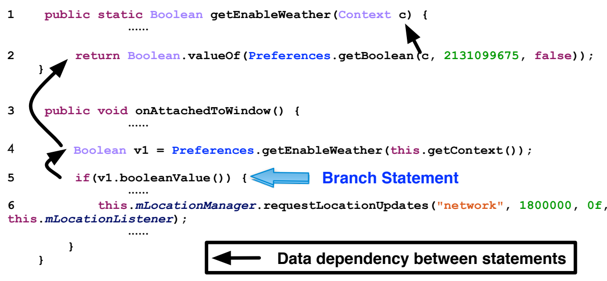

```
1 package com.android.inputmethod.latin.settings;
2 final class t extends AsyncTask {
3    private Integer a() {
4         try {
              ...
5              Iterator v5 =this.a.getActivity().
  getPackageManager().getInstalledPackages(8192).iterator();
6              while(true) {
                  ...
7                  Object v0_1 = v5.next();
                  ...
8                  String v6 = ((PackageInfo)v0_1).packageName;
                  ...
9                  Log.e("package", v6);
                  ...
          }}}}
```

Fig. 6. Code snippet of `com.qisiemoji.inputmethod`: obtains the installed package list and writes it to log.

In detail, for each sensitive API/URI, we first obtain the control flow path from the entry point (e.g., *Activity.onCreate()*, *onClick()*) to it. Then, we extract all the branch statements (i.e., `if_stmt`, `lookup_switch_stmt`, and `table_switch_stmt`) [63] in the control flow path (e.g., Fig. 7 Line 5). To determine the factors that will affect the execution of privacy related behavior, for each branch statement, we traverse the data dependency graph from it to get all the statements related to the branch statement. In data flow analysis [64], if a variable v is on the LHS (i.e., left-hand side) of an assignment statement $s(j)$, the statement $s(j)$ is a *definition* of variable v. For example, in Fig. 7, variable $v1$ is defined in Line 4 and this statement also *use* the return value of the method *getEnableWeather()*. If the traversal reaches a statement that does not *use* any pre-defined variables, the traversal stops. In Fig.7, Line 2 and Line 4 are extracted if we start traversal from the branch statement (i.e., Line 5).



```
1    public static Boolean getEnableWeather(Context c) {
              ......
2        return Boolean.valueOf(Preferences.getBoolean(c, 2131099675, false));
     }

3    public void onAttachedToWindow() {
              ......
4        Boolean v1 = Preferences.getEnableWeather(this.getContext());
5        if(v1.booleanValue()) {
              ......
6            this.mLocationManager.requestLocationUpdates("network", 1800000, 0f,
this.mLocationListener);
              ......
         }
     }
```

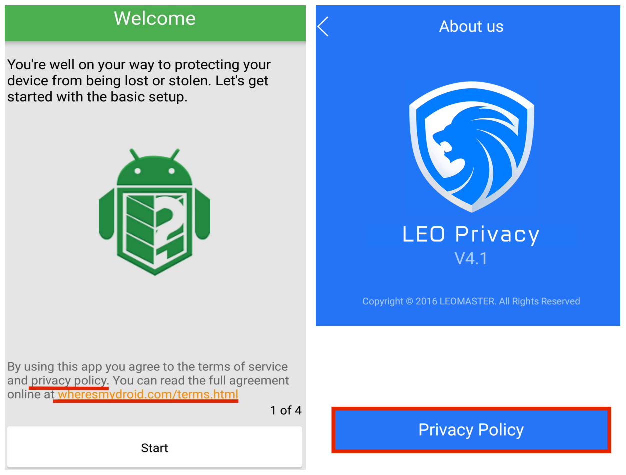
Fig. 7. Code snippet of `com.media1908.lightningbug`: Extracting the constraints.

6. Locating the In-App Privacy Policy

The in-app privacy policy is usually provided in two ways: (1) showing the URL of privacy policy on the GUI (e.g., the left GUI in Fig. 8); (2) displaying the privacy policy after pressing a button (e.g., right GUI in Fig. 8). To collect such in-app privacy policy, we first parse the layout file of each activity to get the GUI structure. Then, for the first scenario, we check whether the text on the GUI contains "privacy policy". If so, we extract the URLs contained in the HTML attribute <*a href*="" > and download the content. For the second scenario, if the widget (e.g., button) is associated with the text "privacy policy", we check the resource ID and use it to find the callbacks registered in code. Then, we traverse the control flow graph from the entry of the callback to find the statement that loads HTML page (e.g., *WebView.loadUrl()*). To get the URL of privacy policy, for these statements, we leverage constant propagation technique [57] to identify all string parameters that start with "http"/"https". After that, we download the content through these URLs.

7. Correlating UI Elements and Texts The developers usually provide UI (e.g., activity, dialog) to let user enable feature/function/service. To identify which feature/function/service is enabled by user, the static analysis module will create a mapping between the resource IDs of UI elements and their texts describing the enabled feature/function/service.

When designing the layout of an activity, the developer may use various UI elements to support user selection (e.g., *CompoundButton*, *CheckBox*, *RadioButton*, *Switch*, *ToggleButton*). We parse the layout XML file to get the resource ids of these UI elements. If the developer embeds descriptive text as attributes (e.g., `android:text`) of the UI element, we extract the value of these attributes to get the related string (i.e., text). Otherwise, we follow the method in [65] to locate nearby text widget in UI and extract its descriptive text. We regard this text as the text related to the UI element. For pop-up dialog [66], since developer can also set its layout via *setContentView()* (similar to *Activity* class), we analyze its layout and map the UI element to text label. Different from activity and dialog, the layout of preference is defined in the `res/xml` folder [67]. We parse the XML file to get the resource ids of the UI elements. For each UI element, we map its resource ID to the string defined in `android:title` attribute. For example, the preference activity of app (`com.media1908.lightningbug`) include a *CheckBoxPreference* widget. The id of this widget is `0x7f06001b` and the value of the string `android:title` attribute is string "*Local Weather*". Thus, we map the resource ID `0x7f06001b` to this string.



Welcome

You're well on your way to protecting your device from being lost or stolen. Let's get started with the basic setup.

By using this app you agree to the terms of service and privacy policy. You can read the full agreement online at wheresmydroid.com/terms.html

1 of 4

Start

About us

LEO Privacy
V4.1

Copyright © 2016 LEOMASTER. All Rights Reserved

Privacy Policy

Fig. 8. GUI of In-App Privacy Policy.

D. Description and What's New Analysis Module

Table V describes the symbols representing the information extracted from app description and what's new.

TABLE V
MAJOR SYMBOLS RELATED TO DESCRIPTION AND WHAT'S NEW.

Symbol	Meaning
Res^{Desc}	the set of resources used by app according to description
$Res^{Whatsnew}$	the set of resources used by app according to what's new

Description analysis. We improve the state-of-art description analysis system, *AutoCog*, to map an app's description to permissions [45]. *AutoCog* conducts statistical analysis on a large number of apps to build up the description to permission relatedness (DPR) model. More precisely, it extracts (verb, noun) pairs (e.g., ("track", "location")) from description, and maps them to different permissions (e.g., `ACCESS_FINE_LOCATION`). We map the permissions inferred from description to personal information by analyzing the official document. For example, permission `ACCESS_FINE_LOCATION` is mapped to "location", "latitude", and "longitude" since the APIs *getLastKnownLocation()*, *getLatitude()*, and *getLongitude()* require this permission. Let Res_{Desc} denote the collected information that is inferred from the app's descriptions.

We find that *AutoCog* may generate false negatives (i.e., miss the mapping between description and permission) due to two reasons. First, some privacy related verb phrases are not included in the DPR module of *AutoCog*. An example is the sentence "*You can use GPS on your mobile device ...*". Since the verb phrase ("use", "GPS") is not included in the DPR model of the permission `ACCESS_FINE_LOCATION`, *AutoCog* cannot infer from this sentence that the app uses location related permission. Second, the DPR model of *AutoCog* focuses on verb phrases and it will not analyze the noun phrases followed by the preposition "*with/to/from*". For instance, in the sentence "*Connect LIVE with our totally unique voice feature*", the noun phrase "voice feature" is ignored by *AutoCog* and thus it will not be mapped to the `RECORD_AUDIO` permission.

To mitigate the first kind of false negatives caused by the incompleteness of DPR model, for each permission, we generate a series of new verb phrases and add them to the DPR model. The verbs of these new verb phrases are the verbs listed in Tab. I. The nouns of these new verb phrases are the personal information protected by each permission. To avoid the second kind of false negatives caused by the noun phrases ignored by *AutoCog*, we first extract the noun phrases followed by the preposition "*with/to/from*" from the parse tree of the description sentence. Then, we use ESA [44] to calculate the semantic similarity between these noun phrases and the personal information protected by each permission. If the similarity exceeds a threshold, we map the description sentence to the corresponding permission.

What's new analysis. Since the sentences in WsN are similar to that in the description, we employ the same method for description analysis to map WsN to different kinds of permissions/personal information. However, WsN contains many short sentences that only have noun phrase. For example, "*New effects*". When identifying the permissions mentioned in WsN, the description analysis module will not analyze them since they do not contain any verb phrases.

To avoid such false negatives, we enhance WsN analysis by extracting all the noun phrases from the parse tree. Then, we use ESA [44] to calculate the semantic similarity between these noun phrases and the personal information protected by permission. If the similarity reaches a pre-defined threshold, we think the corresponding permission is mentioned in the WsN. For example, we map the noun phrase "*SD Card Backup*" to permission `WRITE_EXTERNAL_STORAGE` since it is similar to "SD Card". Let $Res^{Whatsnew}$ denote the collected information inferred from the app's WsN.

IV. Problem Identification Module

This section describes how to detect the issues in privacy policy, namely, incomplete privacy policy (Section IV-A), incorrect privacy policy(Section IV-B), imprecise privacy policy (Section IV-C), inconsistent privacy policy (Section IV-D), and user-unfriendly privacy policy (Section IV-E). Section IV-F describes the steps for examining the third-party libs' privacy policies.

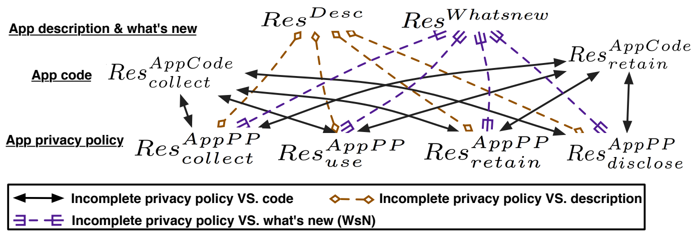

Res^{Desc}
$Res^{Whatsnew}$
App description & what's new
App code
$Res^{AppCode}_{collect}$
$Res^{AppCode}_{retain}$
App privacy policy
$Res^{AppPP}_{collect}$
Res^{AppPP}_{use}
Res^{AppPP}_{retain}
$Res^{AppPP}_{disclose}$
Incomplete privacy policy VS. code
Incomplete privacy policy VS. description
Incomplete privacy policy VS. what's new (WsN)

Fig. 9. Model of incomplete privacy policy.

A. Detecting Incomplete Privacy Policy

Fig.9 shows how to identify an incomplete privacy policy through two general approaches. First, if the information listed in a privacy policy cannot cover that inferred from the description or WsN, the privacy policy is incomplete. Second, if the information declared in a privacy policy cannot cover the collected or retained information which is determined from the corresponding app's bytecode, the privacy policy is incomplete.

Algorithm 1: Detect Incomplete Privacy Policy through Description

```
Input: $Res^{AppPP}_{collect}$, $Res^{AppPP}_{use}$, $Res^{AppPP}_{retain}$, $Res^{AppPP}_{disclose}$: information
       collected, used, retained or disclosed by app privacy policy; $Res^{Desc}$:
       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  $PPInfos = Res^{AppPP}_{collect} \cup Res^{AppPP}_{use} \cup Res^{AppPP}_{retain} \cup Res^{AppPP}_{disclose}$;
2  $CodeInfos = Res^{Desc}$;
3  $ProblemInfos = []$;
4  for $Info$ in $CodeInfos$ do
5      $FindSimilarInfo = 0$;
6      for $PPInfo$ in $PPInfos$ do
7          if Similarity($Info$, $PPInfo$) > threshold then
8              $FindSimilarInfo = 1$;
9          end
10     end
11     if $FindSimilarInfo == 0$ then
12         $ProblemInfos$.append($Info$); //Save the missed Info
13     end
14 end
15 if $ProblemInfos$.length() > 0 then
16     return $ProblemInfos$; //Privacy policy is incomplete
17 end
18 return Null; //Privacy policy is not incomplete
```

Detecting incomplete privacy policy through description. Algorithm 1 shows the process of detecting incomplete privacy policy through description. The description analysis module provides the information used by an app (i.e., Res^{Desc}) while the privacy policy analysis module lists the information to be collected/used/retained/disclosed by the app (i.e., $PPInfos$, line 1). We compare each information in Res^{Desc} with all the information identified from the privacy policy (line 6-10). If no private information pairs can be matched, the $Info$ is missed by the privacy policy and hence the privacy policy is incomplete. Here, "match" means that two kinds of information refer to the same thing.

We use ESA [44] to measure the semantic similarity between two information. If the similarity is larger than a threshold, we regard them as the same thing (line 7 in Algorithm 1). By default, the threshold is 0.67 following [45].

Detecting incomplete privacy policy through what's new. The incomplete privacy policy appears if the information described in WsN (i.e., $Res^{Whatsnew}$) is not mentioned in the privacy policy. To detect such privacy policy, we reuse Algorithm 1 but replace $CodeInfos$ with $Res^{Whatsnew}$.

Detecting incomplete privacy policy through code. The static analysis module outputs the information collected or retained. We compare each information with all the information identified from the privacy policy. If the privacy policy misses any such information, it is incomplete. The algorithm is similar to Algorithm 1. The only difference is that we replace $CodeInfos$ with $Res^{AppCode}_{collect} \cup Res^{AppCode}_{retain}$. Since some information requires certain permission (e.g., account requires `GET_ACCOUNTS`), in this case, we only consider the app that requires the corresponding permissions.

For the incomplete privacy policy detected through description, WsN, and code, we will further check the information that the app's privacy policy declares to collect, use, retain, disclose (i.e., $Res^{AppPP}_{collect}$, Res^{AppPP}_{use}, Res^{AppPP}_{retain}, $Res^{AppPP}_{disclose}$). If the incomplete privacy policy does not mention that the general information (i.e., "*personal information*", "*personal data*", "*personally identifiable information*") will be collected, used, retained, or disclosed by the app, we will report it as an incomplete privacy policy. Otherwise, we regard it as an imprecise privacy policy and the corresponding detection is described in Section IV-C).

B. Discovering Incorrect Privacy Policy

Incorrect privacy policy declares not to access certain personal information but the corresponding apps do it. As shown in Fig.10, we detect the incorrect privacy through two approaches. First, if the privacy policy declares not to collect personal information whereas the permissions inferred from the description or WsN cover such information, the privacy policy is incorrect. Second, if the privacy policy declares not to collect or retain certain personal information whereas such behavior is realized in code, the privacy policy is incorrect. Since it is difficult to differentiate between the *collected* resources and the *used* resources according to the code, we contrast both $\overline{Res^{AppPP}_{collect}}$ and $\overline{Res^{AppPP}_{use}}$ with $Res^{AppCode}_{collect}$.

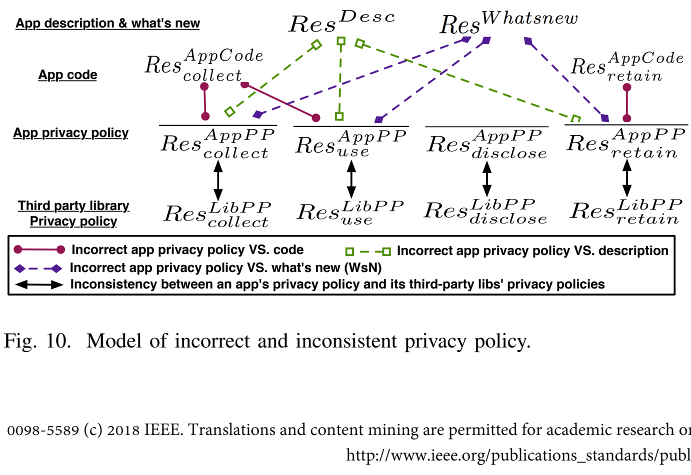

App description & what's new
Res^{Desc}
$Res^{Whatsnew}$
App code
$Res^{AppCode}_{collect}$
$Res^{AppCode}_{retain}$
App privacy policy
$\overline{Res^{AppPP}_{collect}}$
$\overline{Res^{AppPP}_{use}}$
$\overline{Res^{AppPP}_{disclose}}$
$\overline{Res^{AppPP}_{retain}}$
Third party library Privacy policy
$Res^{LibPP}_{collect}$
Res^{LibPP}_{use}
$Res^{LibPP}_{disclose}$
Res^{LibPP}_{retain}
Incorrect app privacy policy VS. code
Incorrect app privacy policy VS. description
Incorrect app privacy policy VS. what's new (WsN)
Inconsistency between an app's privacy policy and its third-party libs' privacy policies

Fig. 10. Model of incorrect and inconsistent privacy policy.

Discovering incorrect privacy policy through app's description. Using an app's description can discover two kinds of incorrect privacy policy. First, the privacy policy declares not to collect or use certain personal information, but the description indicates that the corresponding permissions are required by the app. For each information to be collected, used, retained, or disclosed in Res^{Desc}, we compare it with the personal information that the app's privacy policy declares *not* to collect or use. If a pair of such information is found, the privacy policy is incorrect. The algorithm of detecting incorrect privacy policy through description is similar to Algorithm 1. The only difference is that we use $\overline{Res^{AppPP}_{collect}} \cup \overline{Res^{AppPP}_{use}}$ to replace $PPInfos$. Second, the privacy policy declares not to access *any* personal information, but the description indicates that the app needs some permissions required for accessing certain personal information. To detect such privacy policy, we first identify all privacy policies that declare not to collect or use "any personal information" or "any personal data", and then check the corresponding permissions inferred from the descriptions. If an app requests any sensitive permission, the privacy policy is incorrect.

Discovering incorrect privacy policy through what's new. The procedure of discovering incorrect privacy policy through WsN is similar to Algorithm 1. Specially, we use $Res^{Whatsnew}$ to replace $CodeInfos$ and use $\overline{Res^{AppPP}_{collect}} \cup \overline{Res^{AppPP}_{use}}$ to replace $PPInfos$.

Discovering incorrect privacy policy through code *PPChecker* can detect incorrect privacy policy that declares not to collect/use certain information but the app code does. The corresponding algorithm is similar to Algorithm 1. The only difference is that we use $\overline{Res^{AppPP}_{collect}} \cup \overline{Res^{AppPP}_{use}}$ to replace $PPInfos$. We also utilize $Res^{AppCode}_{collect}$ to replace $CodeInfos$. Similarly, *PPChecker* can detect incorrect privacy policy that declares not to retain certain information but the app code does. The corresponding algorithm is similar to Algorithm 1. The only difference is that we use $\overline{Res^{AppPP}_{retain}}$ and $Res^{AppCode}_{retain}$ to replace $PPInfos$ and $CodeInfos$, respectively.

C. Identifying Imprecise Privacy Policy

Discovering the first kind of imprecise privacy policy. The first kind of imprecise privacy policy does not clearly specify the personal information accessed by the app. To detect such imprecise privacy policy, we first identify the privacy policy that does not specify the personal information used in description/WsN/code. Then, we check whether or not the privacy policy mentions some abstraction of the personal information. If yes, the privacy policy is imprecise. Otherwise, it is an incomplete privacy policy (examined in Section IV-A). The detection procedure includes two steps:

Step 1: To identify the privacy policy that cannot cover the personal information used in description, WsN, and code, we reuse the incomplete privacy policy detection algorithm (Algorithm 1) by replacing $CodeInfos$ with Res^{Desc}, $Res^{Whatsnew}$, and $Res^{AppCode}_{collect} \cup Res^{AppCode}_{retain}$, respectively.

Step 2: If the privacy policy cannot cover the information identified from the description or the code, we check whether the privacy policy mentions the abstraction of the personal information. If so, this privacy policy is imprecise because it does not list the specific personal information to be accessed. For example, if the app's code collects "phone number" but its privacy policy only declares that it will access "personal information", the privacy policy is imprecise.

Breaux et al. created the abstraction of personal information by manually reading three privacy policies and identifying the statement that contains a definition or elaboration of a phrase-level concept [42]. Then, they transform such statement into Eddy syntax. Three kinds of statements are analyzed: (1) *refinement*, which means that one concept is refined by a more specific concept; (2) *abstraction*, which means that a list of concept is described by a more general concept; (3) *exclusion*, which means that a concept is excluded from another concept.

We follow this procedure to collect the abstraction of personal information. Unfortunately, we find that the abstraction contained in different apps' privacy policies may not be consistent. For example, the privacy policy of `com.absi.tfctv` declares that "*non-personal information that may include your IP address*" while the privacy policy of `com.fincon.globalHH` describes that "*collect your personal information, such as your IP address*". Hence, we cannot use a few privacy policies to create the abstraction for *PPChecker*. Instead, to avoid such inconsistency, we select the data definition in GDPR [68] and the privacy policy of Google Play [69] to create the abstraction. For example, the sentence "*a unique identifier such as the Advertising ID is used to ...*" refines the concept "unique identifier" with the specific concept "Advertising ID". We infer that "unique identifier" is an abstraction of "Advertising ID".

Discovering the second kind of imprecise privacy policy. The second kind of imprecise privacy policy declares some privacy related behaviors that do not exist in code. We identify them by comparing the personal information declared in privacy policy with the personal information accessed in code. In detail, for each information that the app privacy policy declares to collect/use/retain the resource (i.e., $Res^{AppPP}_{collect}$, Res^{AppPP}_{use}, Res^{AppPP}_{retain}), we use ESA [44] to check if it is similar to one of the 14 kinds of personal information that can be checked by the static analysis module (Section III-C 2. Identifying the Collected Information). If true, we check whether the corresponding APIs/URIs are used in code or not. If we cannot find any related APIs/URIs in code, the privacy policy is imprecise.

Discovering the third kind of imprecise privacy policy. The third kind of imprecise privacy policy uses the adverb clause of condition to describe the condition of privacy related behavior, but the code does not check whether the condition is satisfied or not. We consider two types of adverb clause of conditions (Tab. III). **Type 1:** Adverb clause of conditions related to enabling service. If the privacy policy describes that personal information is accessed when a certain service is enabled, the code must check if the corresponding service is enabled or not before accessing the personal information. Otherwise, the privacy policy is imprecise. Here we consider both the system services and the function/feature implemented by the developers. **Type 2:** Adverb clause of conditions related to UI callback. If the privacy policy declares that personal information will be accessed after pressing UI widget, we check if the code access the corresponding personal information in the UI callback or not. If the personal information is not accessed in UI callback, the privacy policy is imprecise.

To detect this kind of imprecise privacy policy, after extracting (verb, noun) pairs from the typed dependency relations of the adverb clause of condition, we use these (verb, noun) pairs to determine the type of constraints and then conduct the checking. The procedures are detailed as follows.

- **Type 1: Adverb clause of conditions related to enabling service.** If the (verb, noun) pair is similar to ("enable", "service/function/feature"), we regard this constraint as the first type of constraint, and extract the noun of the (verb, noun) pair. We first check whether the feature/function/service is a system service or not by comparing the noun of the (verb, noun) pair with the 46 kinds of system service (e.g., "location", "bluetooth") listed in [70]. If we cannot find similar system service, the adverb clause of condition may be related to a function/feature implemented by the developer, and therefore we compare the service/function/feature mentioned in adverb clause of conditions with the constraints extracted from code.

If the adverb clause of condition is related to system service, we check whether the app code has called the corresponding APIs or not before accessing sensitive information. For example, in the constraint "*if you enable location service*", since "location service" is a system service, the API *LocationManager.isProviderEnabled()* should be called before accessing the information. If the app does not call this API, the adverb clause of condition is imprecise.

If the adverb clause of condition is related to the function/feature implemented by developer, we first get the resource ID of the UI element that allows user to enable it, and then we check whether the resource ID is used in the constraint of privacy-related behavior or not. In detail, the app usually provides a configuration activity or pop-up dialog to allow user to enable function/feature, and the app can learn the user's selection through the status of UI elements (e.g., *CompoundButton.isChecked()*). To obtain the corresponding resource id, we first search the feature/function mentioned in the mapping between the resource id and the text (Section III-C 7), and then check if the constraint of sensitive API/URI (identified in Section III-C 5) contains the resource ID. If yes, the app has checked the status of feature/function in code before accessing personal information. Otherwise, the adverb clause of condition is imprecise.

For example, the privacy policy of the app `com.media1908.lightningbug` contains the sentence "*If you do enable the local weather feature, your approximate location is collected on a periodic basis*". We extract the ("enable", "local weather feature") from its typed dependency relations. "local weather" is a function implemented by developer instead of a system service. We search it in the mapping between resource ID and text. As there is a mapping between `0x7f06001b` and "*Local Weather*", the status of the UI element with resource ID `0x7f06001b` should be checked before the app accesses the location information. As shown in Fig. 7, in the constraint of location related API *requestLocationUpdate()*, we can find that the resource ID `0x7f06001b` (i.e., 2131099675) is used in the constraint. In other words, the app has checked whether users have enabled the feature/function before accessing information.

• **Type 2: Adverb clause of conditions related to UI callback.** If the (verb, noun) pair extracted from the adverb clause of condition has similar meaning as ("press/click", *object*), it is regarded as the second type of constraint. We extract the pressed/clicked *object* based on the typed dependency relations, and then locate the UI element related to the *object*. Finally, we check whether the sensitive information is accessed after pressing the corresponding UI element.

In detail, to get the resource id of the pressed/clicked UI element, we search the extracted object in the mapping between resource ID and text (Section III-C 7). After finding the resource ID of the UI element, we check if the sensitive information is accessed in the callback of the UI element or other activities launched by the UI element. For the former case, we look for the control flow path that starts from the entry of the callback and ends at the invocation of sensitive API/URI. If such path is found, the sensitive information is accessed in the callback. For the latter case, we generate the windows transaction graph (WTG) using Gator [71], and collect the activities that will be launched after pressing the UI element. We scan the code of these activities to determine whether they access the corresponding sensitive information or not. If the information is not accessed in the callback of UI element or the activities launched by the UI element, the adverb clause of condition is imprecise.

D. Revealing Inconsistent Privacy Policy

Inconsistency between app's privacy policy and third-party lib's privacy policy. Fig.10 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 three conditions 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; and
(3) $AppSent_i$ and $LibSent_j$ refer to the same resource.

Algorithm 2 lists the detailed steps. Line 4-5 are related to conditions (1) and (2), where the function $getVerbCategory()$ returns the category of a sentence's main verb and the function $IsPositive()$ returns true if a sentence is positive. Line 6-14 are related to condition (3), where the function $getRes()$ returns the resources extracted from a sentence and the function $Similarity()$ computes the similarity between two resources by using ESA [44]. Some apps' privacy policies declare that they are not responsible for the behaviors of third-party libs. In this case, if the app's privacy policy is inconsistent with third-party libs' privacy policies, we ignore such inconsistency.

Algorithm 2: Reveal Inconsistency between an app's privacy policy and its third-party libs' privacy policies

```
Input: $AppSent_i$ ($1 \leq i \leq m$): Sentences of app privacy policy, $LibSent_j$
       ($1 \leq j \leq n$): Sentences of lib 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
```

Inconsistency between app's privacy policy and in-app privacy policy. After extracting the in-app privacy policy(Section III-C), we use the privacy policy analysis module (Section III-B) to process it and get the information collected/used/retained/disclosed by the app. We compare them with the information obtained from the app's privacy policy (i.e., privacy policy link on Google Play). If one information is collected/used/retained/disclosed in one privacy policy, but it is not mentioned in the other privacy policy, we regard that the app's privacy policy (i.e., online privacy policy) is inconsistent with the in-app privacy policy.

E. Identifying User Unfriendly Privacy Policy

Discovering the language inconsistency between the description and privacy policy. Since we currently just examine the descriptions in English, if the privacy policy is in other language, we report that the privacy policy is not user friendly. To detect whether the privacy policy is written in other language, *PPChecker* utilizes Compact Language Detector [72] to get the top three languages found and their approximate percentages. Compact Language Detector uses Naive Bayesian classifier to detect more than 83 languages (including those languages that contain many ASCII characters, such as French, German, and Italian). If more than 70% of the content is not in English, we regard that the privacy policy is not in English.

Evaluating the readability of a privacy policy. Flesch Reading Ease score [73] uses the average sentence length and word length to measure the readability of the text. Text with a low score is difficult to read. Therefore, we calculate the Flesch Reading Ease score of each privacy policy. Tab. VI shows the meaning of different scores [74]. If the score is lower than 30.0, the text is very difficult to read.

TABLE VI
THE FLESCH READING EASE SCORE OF THE APPS' PRIVACY POLICIES.

Score	School Level	Meaning	# Apps
100.0 - 90.0	5th grade	Very easy to read.	7
90.0 - 80.0	6th grade	Easy to read.	3
80.0 - 70.0	7th grade	Fairly easy to read.	27
70.0 - 60.0	8th and 9th grade	Plain English.	106
60.0 - 50.0	10th to 12th grade	Fairly difficult to read.	187
50.0 - 30.0	College	Difficult to read.	1174
30.0 - 0.0	College graduate	Very Difficult to read.	443

F. Disclosing Problems in Third-party Libs' Privacy Policies

As a third-party lib may also have privacy policy, *PPChecker* also checks whether it is incomplete and/or incorrect. To detect the incomplete privacy policy of third-party lib, we adopt the algorithm of detecting incomplete privacy policy through code (Algorithm 1). In detail, we replace $PPInfos$ with the behaviors declared in a lib's privacy policy ($Res_{collect}^{LibPP}$, Res_{use}^{LibPP}, Res_{retain}^{LibPP}, and $Res_{disclose}^{LibPP}$), and replace the $CodeInfos$ with the behaviors in an lib's code ($Res_{collect}^{LibCode}$, $Res_{retain}^{LibCode}$).

Similarly, to detect the incorrect lib privacy policy, we customize Algorithm 1 by replacing $CodeInfos$ with the information retained by a lib's code ($Res_{retain}^{LibCode}$). We also replace the $PPInfos$ with the information that a lib's privacy policy declare not to retain ($\overline{Res_{retain}^{LibPP}}$).

V. Experimental Result

A. Data Set

We download 2,500 apps from Google Play, each of which has a description in English and provides a privacy policy link. We also examine the privacy policies of three kinds of third-party libs, including:

(1) Ad libs. 52 out of top 90 popular Ad libs in [75] are selected because they have privacy policies in English.

(2) Social libs. 9 out of 18 most popular social network libs in [76] are chosen because they offer privacy policies in English.

(3) Development tools. We pick 20 most commonly used development tools with privacy policies in English from [77] because the majority of other tools do not have websites showing their privacy policies.

In this section, we use experiments to answer the following questions:

Q1: How is the precision and recall rate of privacy policy analysis module ? (Section V-B)

Q2: How many incomplete privacy policies can be detected by *PPChecker* ? (Section V-C)

Q3: How many incorrect privacy policies can be found by *PPChecker* ? (Section V-D)

Q4: How many imprecise privacy policies can be identified by *PPChecker* ? (Section V-E)

Q5: How many inconsistent privacy policies can be revealed by *PPChecker* ? (Section V-F)

Q6: How many user-unfriendly privacy policies can be reported by *PPChecker* ? (Section V-G)

Q7: How many problematic lib privacy policies can be discovered by *PPChecker* ? (Section V-H)

B. Correctness of Sentence Identification in Privacy Policies

To measure the precision and recall rate of the privacy policy analysis module. We manually read 100 privacy policies of mobile apps and create a data set that contains 1000 collection, 1000 usage, 400 retention, and 600 disclosure related sentences. If the sentence is related to the information collected, used, retained, or disclosed by the app, we add "#COLLECT", "#USE", "#RETAIN", or "#DISCLOSE" label to it, respectively.

We compare the new classifier-based sentence identification approach and the previous one in the conference version [34], which uses the patterns learned from privacy policy corpus to identify information collection, usage, retention, and disclosure related sentences. For this evaluation, we randomly select 800 sentences from our privacy policy data set. Half of the sentences are about information collection, usage, retention, and disclosure. The performance of the patterns learned from privacy corpus [34] is shown in Table VII.

TABLE VII
PATTERNS LEARNED FROM PRIVACY CORPUS [34]: IDENTIFYING INFORMATION COLLECTION, USAGE, RETENTION, AND DISCLOSURE RELATED SENTENCES

Sentence Type	Precision	Recall	F-1
Collection, Usage, Retention, Disclosure	78.7%	81.6%	80.1%

From the result shown in Table VII, we can find that due to the limited number of patterns learned from privacy policy corpus, [34] can only achieve the 78.7% precision and 81.6% recall rate.

We select four kinds of commonly used classifiers to test the performance of *PPChecker* when detecting different kinds of sentences, including: Max Entropy [36], SVM [37], naive bayes [38], and Random Forest [39]. We perform 10-fold cross validation on the data set. The result is shown in Table VIII. We select Max Entropy classifier to extract information collection, usage, retention, and disclosure related sentences since its F-1 scores are higher than other three classifiers. The performance is also higher than the patterns learned from privacy corpus [34] (i.e., Table VII).

False positives. The false positive is generated because some sentences contain information collection, usage, retention, or disclosure related verbs, but the object is not personal information. For example, "*please ask your legal guardians permission to use or access the apps*". Although the sentence contains verb "access", its object is "apps". To remove such false positives, we can build up a object blacklist to filter out useless ones.

False negatives. The false negative is caused because some sentences' structures are not included in the training set. For example, the sentence "*We also need a unique ID of your device*" is missed when detecting information collection related

TABLE VIII
REVIEWSOLVER: IDENTIFYING INFORMATION COLLECTION, USAGE, RETENTION, AND DISCLOSURE RELATED SENTENCES

Sentence Type	Classifier	Precision	Recall	F-1
Collection	Max Entropy	98.4%	95.0%	**96.6%**
	SVM	92.2%	84.9%	88.3%
	Naive Bayes	88.7%	91.4%	90.0%
	Random Forest	94.3%	83.6%	88.6%
Usage	Max Entropy	96.5%	90.1%	**93.2%**
	SVM	87.8%	77.3%	82.2%
	Naive Bayes	83.2%	85.9%	84.3%
	Random Forest	86.0%	81.9%	83.7%
Retention	Max Entropy	99.7%	92.7%	**96.0%**
	SVM	95.2%	83.5%	88.7%
	Naive Bayes	94.9%	87.9%	91.2%
	Random Forest	97.1%	79.9%	87.5%
Disclosure	Max Entropy	98.7%	87.9%	**92.9%**
	SVM	94.8%	81.1%	87.2%
	Naive Bayes	87.8%	84.3%	85.8%
	Random Forest	96.0%	77.5%	85.6%

sentences since the training set does not contain sentences with similar structures. To remove such false negatives, we can expand the size of training set to achieve higher coverage.

Answer to Q1: The experimental result shows that: *PPChecker* can achieve at least 96.5% precision and 87.9% recall rate when identifying information collection, usage, retention, and disclosure related sentences.

C. Detecting Incomplete Privacy Policy

Detecting incomplete privacy policy through description. Contrasting an app's description and its privacy policy, *PPChecker* finds 352 questionable apps. Tab. IX 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. Note that we only consider the permissions requested in the manifest file. The results show that many apps need to read and store information in external storage (`WRITE_EXTERNAL_STORAGE`), but not all of them explain this behavior in their privacy policy. These apps may use external storage as cache or store cookie. The location related permissions (i.e., `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`) also affect many apps. Many of these apps belong to the category of weather and the category of map, and they need the location information to provide services. Moreover, the permissions `READ_CONTACTS` and `GET_ACCOUNTS` also affect many apps.

False positives. We randomly select and manually check 20 incomplete privacy policies detected by *PPChecker*. The result shows that all 35 missed permissions are not described in these privacy policies. In other words, no false positives are reported.

False negatives. To measure the false negatives of *PPChecker*, we randomly select 20 privacy policies that are not reported to be incomplete. We manually read the corresponding descriptions and do not find any missed alerts (i.e., the description describes one permission but privacy policy does not mention it).

TABLE IX
PERMISSIONS LEADING TO INCOMPLETE PRIVACY POLICY AND THE NUMBER OF CORRESPONDING APPS (FOUND THROUGH DESCRIPTION).

Permission	Num. of Questionable apps
ACCESS_COARSE_LOCATION	76
ACCESS_FINE_LOCATION	149
CAMERA	30
GET_ACCOUNTS	25
READ_CALENDAR	64
READ_CONTACTS	51
WRITE_CONTACTS	8
RECORD_AUDIO	19
WRITE_SETTINGS	12
WRITE_EXTERNAL_STORAGE	133

Detecting incomplete privacy policy through what's new. *PPChecker* finds 14 incomplete privacy policies through WsN, as shown in Tab. X, which have also been manually verified.

TABLE X
PERMISSIONS LEADING TO INCOMPLETE PRIVACY POLICY AND THE NUMBER OF CORRESPONDING APPS (FOUND THROUGH WHAT'S NEW).

Permission	Num. of Questionable apps
ACCESS_COARSE_LOCATION	1
ACCESS_FINE_LOCATION	4
CAMERA	4
GET_ACCOUNTS	1
READ_CALENDAR	1
READ_CONTACTS	1
WRITE_CONTACTS	1
RECORD_AUDIO	1
WRITE_SETTINGS	1
WRITE_EXTERNAL_STORAGE	3

Detecting incomplete privacy policy through code. By analyzing apps' code, *PPChecker* finds 486 pieces of missed personal information. Manually checking confirms that 435 pieces of missed personal information distributed in 295 privacy policies do have the problem whereas 51 missed personal information are false positives. Among these 295 incomplete privacy policies, we find that 93 pieces of missed personal information are retained.

Fig. 11 lists the distribution of missed information. We can see that the location is the most common information accessed but not mentioned in the incomplete privacy policies. This result is consistent with the result in Tab. IX. We also check the corresponding APIs used by those apps with incomplete privacy policy and find 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 find that the false positives are caused by errors in extracting personal 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 extract "name" since it is the object of the action "collect", but fail to extract "device identifier".

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

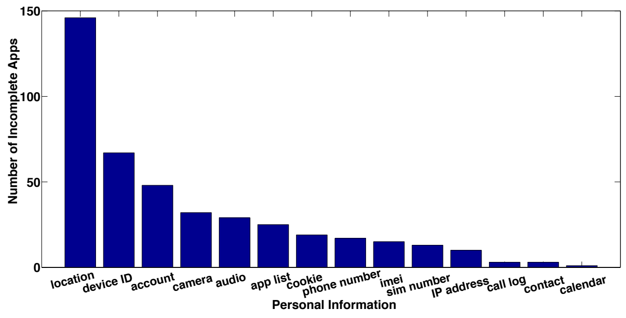


Fig. 11. Distribution of personal information collected or retained by apps with incomplete privacy policies.

Answer to Q2: The experimental result shows that: *PPChecker* detects 492 apps that have incomplete privacy policies: 352 are detected through descriptions, 14 are detected through WsN, and 295 are detected through code.

D. Discovering Incorrect Privacy Policy

Discovering incorrect privacy policy through description. *PPChecker* finds 2 apps having the first kind of incorrect privacy policy, namely, `com.humetrix.icebb` and `com.herman.ringtone`. The former's privacy policy says *"The App does **not** collect information about the location of a users mobile device."*. But its description describes the use of location, *"ICEBlueButton automatically generates ... a map with the location of the emergency"* and the app requests the permission `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. The latter's privacy policy declares *"Ringtone Maker will **not** collect your contact information"*. But its description describes the use of contact information, *"After you create the ringtone, there is a choice to assign it to your contact"*, and the app requests the permission `WRITE_CONTACTS`. Manual checking shows that this app provides an activity for assigning ringtone to contact: (1) After pressing the "Contact" button, it reads the whole contact list stored in your device and displays it on the GUI. (2) If the user clicks one contact item, it pop-ups a window for ringtone selection. *PPChecker* also detects 6 apps having the second kind of incorrect privacy policies. For example, the app's (`eu.asteryx.transportlocator`) privacy policy declares *"TMUApps does **not** collect or store any personal information through any of its apps."*. But it requests the permission `ACCESS_FINE_LOCATION` to provide map service.
Discovering incorrect privacy policy through what's new. Since apps only have a few sentences in WsN, *PPChecker* does not find any incorrect privacy policy through WsN.
Discovering incorrect privacy policy through code. *PPChecker* finds one app with incorrect privacy policy by comparing $\overline{Res^{AppPP}_{collect} \cup Res^{AppPP}_{use}}$ and $Res^{AppCode}_{collect}$. Although `com.humetrix.icebb` declares not to collect location information in their privacy policies, it calls the API *Location.getLatitude()* and *Location.getLongitude()* in code.

By comparing $\overline{Res^{AppPP}_{restain}}$ and $Res^{AppCode}_{retain}$, *PPChecker* finds 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* finds a path from *getLatitude()* to *Log.i()* (i.e., the location information will be stored in log.

False positives. We observe two false positives due to the lack of consideration of the context. For example, *PPChecker* finds from the code of `com.zoho.mail` that it will access the account information. However, it correlates this behavior with the sentence "*We also do not process the contents of your user account for serving targeted advertisements*" mistakenly, and thus raises 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 select 20 apps to check whether *PPChecker* causes any false negatives. The negative sentences in privacy policies, the permissions inferred from descriptions, and the information collected/retained in code are all inspected, and we do not find any false negatives.

Answer to Q3: The experimental result shows that: *PPChecker* discovers 10 apps that have incorrect privacy policies. 8 of them contain conflicts between descriptions and privacy policies. 3 of them contain conflicts between code and privacy policies.

E. Identifying Imprecise Privacy Policy

Discovering the first kind of imprecise privacy policy. *PPChecker* discovers 451 apps that do not clearly describe privacy-related behaviors. In detail, by using the permissions inferred from description, *PPChecker* discovers 238 such imprecise privacy policies. We count the number of questionable apps for each permission. The result is shown in Tab. XI. We can find that `ACCESS_FINE_LOCATION` is the most commonly missed permissions for these apps. By using the permissions inferred from WsN, *PPChecker* discovers 23 imprecise privacy policies. By using the information accessed in code, *PPChecker* discovers 288 apps that access sensitive information in code, but they do not clearly describe these behaviors in privacy policies.

False positives. We randomly select 20 questionable apps and check the corresponding description/WsN/code. All these apps' description/WsN/code contain privacy-related behaviors but the corresponding privacy policies do not clearly describe them. In other words, we do not find any false positives.

False negatives. We randomly select 20 privacy policies that are not reported to be the first kind of imprecise privacy policies. We check if the corresponding description/WsN/code contain privacy-related behavior and the privacy policy does

TABLE XI
PERMISSIONS LEADING TO IMPRECISE PRIVACY POLICY AND THE NUMBER OF CORRESPONDING APPS (FOUND THROUGH DESCRIPTION).

Permission	Num. of Questionable apps
ACCESS_COARSE_LOCATION	55
ACCESS_FINE_LOCATION	77
CAMERA	23
GET_ACCOUNTS	16
READ_CALENDAR	14
READ_CONTACTS	62
WRITE_CONTACTS	11
RECORD_AUDIO	30
WRITE_SETTINGS	14
WRITE_EXTERNAL_STORAGE	56

not clearly describe it. Finally, we do not find any false negatives.

Discovering the second kind of imprecise privacy policy. *PPChecker* discovers 967 imprecise privacy policies that describe privacy-related behaviors, but we cannot locate these behaviors in code. For example, the privacy policy of the app `air.com.disney.hiddendisney.goo` describes that "*when you visit our sites or use our applications, including location information either provided by a mobile device*". However, as the app does not request location related permissions, it cannot call location related APIs in code.

False positives. We randomly select and manually check 20 questionable apps found by *PPChecker*. We discover 2 false positives. These two privacy policies contain sentences related to information query but *PPChecker* regards them as information collection related sentences. For example, the privacy policy of the app `air.com.kitchenscramble.goo` contains the sentence: "*please send a request by email with your current contact information to ...*". The privacy policy analysis module of *PPChecker* extracts the "contact information" from the sentence and sends out an alert.

False negatives. We randomly select 20 privacy policies that are not reported as the second kind of imprecise privacy policies. We check whether these privacy policies contain privacy related behaviors that cannot be located in code. We cannot find any false negatives.

Discovering the third kind of imprecise privacy policy. After checking the adverb clause of condition of privacy policies, *PPChecker* identifies 2 imprecise privacy policies that the adverb clause of condition cannot be found in code. One is related to the first type of constraint shown in Tab. III. The privacy policy of the app `me.fahlo` describes that "*If you choose to enable push notifications, we may use your Personal Information, or nonpersonal information such as a device ID*". The static analysis module found that the app will collect and transmit the device ID when logging in. However, the app will not check whether push notification is enabled in this procedure. In other words, the device ID will be collected automatically. As the adverb clause of condition contained in privacy policy is not implemented in code, the privacy policy is imprecise. The other is related to the second type of constraint shown in Tab. III. The privacy policy of the app *mobisocial.omlet* declares that "*If you click go, we will collect information about your geographic location*". However, in app code, the location-related API (*LocationManager.getLastKnownLocation()*) is called by the *com.baidu.android.pushservice.b.e.h()* method, instead of the button's callback function(i.e., *onClick()*). Since the privacy policy describes that location information will be collected in the UI callback but this condition cannot be found in code, this privacy policy is imprecise.

False positive. We manually check the code of these two apps and verify these two alerts.

False negatives. For the other privacy policies that contain useful sentences with constraints shown in Tab. III, we manually check the code and do not find any missed alerts.

Answer to Q4: The experimental result shows that: *PPChecker* identifies 1259 apps with imprecise privacy policies.

F. Revealing Inconsistent Privacy Policy

Discovering Inconsistency Between App Privacy Policy and Lib Privacy Policy. By comparing the app privacy policies with their related lib privacy policies, the result shows that 153 apps' privacy policies contain inconsistent sentences.

To evaluate the recall rate, we randomly select 200 app from the data set. We first extract all negative sentences related to information collection/usage/retention/disclosure from these apps' privacy policies. Then, we extract all positive sentences related to information collection/usage/retention/disclosure from the corresponding libs' privacy policies. We manually compare these two kinds of sentences and discover 13 inconsistent privacy policies. *PPChecker* detects 11 apps. Hence, the recall rate is 84.6%.

False positives. We find that the false positive is due to the fact that ESA 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 the former to the "personal information" in the latter, and regards them as the same thing by mistake.

False negatives. 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*". The privacy policy analysis module of *PPChecker* fails to identify the sensitive verb "display", and thus it cannot extract "personal information" from this sentence.

Discovering Inconsistency Between App Privacy Policy and In-app Privacy Policy. As described in Section III-C, the in-app privacy policy links can be extracted from the text of GUI or callback functions of buttons. *PPChecker* extracts 39 privacy policy links from GUI texts and 11 privacy policy links from the callback functions of buttons.

After comparing these in-app privacy policies with their corresponding on-line privacy policies, *PPChecker* discovers 5 problematic apps. For example, the on-line privacy policy of the app `com.wendy` (stored in "*http://www.thirdageapps.com/Privacy/*") says "*We may use* *technologies like unique device identifiers to anonymously identify your computer or device so we can deliver a better experience*", which implies that device ID will be collected. Another sentence "*we and our partners may collect, use, and share precise location data*" describes that location will be collected. However, its in-app privacy policy (store in "*http://thirdageapps.com/wendy/site/privacy*") does not mention these two behaviors at all.

False positives. We manually check these 5 problematic apps and do not find any false alerts.

False negatives. We manually check other apps that contain in-app privacy policy links. The content is the same as the content of on-line privacy policies. We do not find false negatives in them.

Answer to Q5: The experimental result shows that: *PPChecker* reveals 158 apps that have inconsistent privacy policies where 153/5 are detected through comparing app privacy policy with lib/in-app privacy policy, respectively.

G. Recognizing User Unfriendly Privacy Policy

PPChecker finds 501 user unfriendly privacy policies. 58 apps' descriptions are in English but their privacy policies are not in English. These results have been manually validated.

We compute the Flesch reading scores of the all privacy policies and find 443 apps' privacy policies are very difficult to read. The score distribution is shown in the last column of Tab. VI. For example, the score of the app's (`com.MobileTreeApp`) privacy policy is 28.47, meaning that it is very difficult to read its privacy policy. After checking this privacy policy, we find that it has many long sentences with more than 40 words.

Answer to Q6: The experimental result shows that: *PPChecker* recognizes 501 apps have user-unfriendly privacy policies: 58 apps' descriptions are in English but their privacy policies are not in English and 443 apps' privacy policies are very difficult to read.

H. Disclosing Problems in Third-party Libs' Privacy Policies

After checking the privacy policies of 52 ad libs, 9 social libs, and 20 development tools described in Section V-A, we find 5 incomplete lib privacy policies. 3 of them miss declaring the use of device ID. 2 of them do not mention the use of IP address. One lib's privacy policy misses mentioning the use of location and sim serial number. For example, the lib `Pontiflex` calls *getLatitide()* and *getLongitude()* to get latitude and longitude, but its privacy policy does not mention such behavior. Moreover, we do not find any third-party lib that declares not to retain some personal information in privacy policy but conducts such behavior in code.

Answer to Q7: The experimental result shows that: *PPChecker* detects 5 libs with incomplete privacy policies.

I. Summary of the Experimental Result

For 2,500 apps, *PPChecker* finds 1,850 (74.0%) apps' privacy policies having at least one problem. We compared the experimental result of current version *PPChecker* with that of conference version. The result is shown in Tab. XII. For the incomplete and incorrect privacy policies, the percentage of problematic privacy policies detected by current version of *PPChecker* is slightly higher than the conference version. For the inconsistent privacy policies, the percentage of problematic privacy policies detected by current version of *PPChecker* is equal to the result of the conference version. For the other two kinds of problematic privacy policies (i.e., imprecise and user unfriendly privacy policies), only the current version can detect it.

TABLE XII
EXPERIMENTAL RESULT COMPARISON: CURRENT VERSION AND THE CONFERENCE VERSION [34].

Problem Category	Conference Version [34]	Current Version
Incomplete privacy policy	18.5% (222/1197)	19.7%(492/2,500)
Incorrect privacy policy	0.3% (4/1197)	0.4%(10/2,500)
Imprecise privacy policy	-	50.4%(1,259/2,500)
Inconsistent privacy policy	6.3%(75/1197)	6.3%(158/2,500)
User unfriendly privacy policy	-	20.0%(501/2,500)

For the 81 libs, *PPChecker* detects 5 problematic privacy policies.

VI. Threat to Validity

Being a 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. Some threat will affect the performance of *PPChecker*.

Internal threats. Some factors in system design will affect the number of questionable privacy policies detected by *PPChecker*. First, when checking the adverb of condition, we only check the condition related to enabled feature/function/service and UI element clicking. In future work, we will create models for recognizing more conditions. Second, due to the limitation of static code analysis, some source-to-sink paths found by *PPChecker* may not be executed. One potential solution is to conduct dynamic analysis for verifying the result of static analysis. Third, developer may use obfuscation techniques to remove the information in class names and method names. We plan to employ the method in [61] to address this issue in future work.

External threat. Some threats in experiment will affect the correctness of the result. First, when checking the performance of privacy policy analysis module, we used a data set that contain 100 privacy policies. We will add more privacy policies to the data set in future. We will also try different kinds of classification algorithms. Second, when checking the incomplete privacy policies detected by *PPChecker*, we only manually check 20 privacy policies to get the recall rate. More privacy policies will be checked in the future.

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. create a set of grammars and use NLP to identify the rule elements in privacy policy [78]. Costante et al. define five patterns and employ the information extraction techniques to discover the information to be collected by websites [21]. *Text2Policy* uses pre-defined patterns to extract access control policies from natural-language project documents and resource-access information from functional requirements [20].

Breaux et al. define a formal language to find conflicts between privacy policies manually [27], [28]. Moreover, Yamada et al. manually look for conflicts among the privacy policies of a few online social networks [79]. The major difference between this paper and theirs is that *PPChecker automatically* discovers the inconsistencies to avoid time-consuming manual inspection. *Privee* combines crowdsourcing and binary classification techniques to examine web privacy policies [80]. 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. Recently, Schaub et al. explore the design space of privacy notice to help developers select the best notice design [81].

B. Android app analysis

CHEX [59] employs static analysis to detect component hijacking vulnerabilities in Android apps. A number of static analysis research has been done on Android apps. *FlowDroid* [55] is a precise context, flow, field, object-sensitive and lifecycle-aware static taint analysis system for Android apps. Lu et al. [57] analyse both used APIs and content providers, and apply the joint flow analysis technique to find more privacy disclosures. *EdgeMiner* [54] 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 [82]. *Whyper* adopts NLP techniques to process an app's description to find out suspicious permissions [83]. *AutoCog* creates a semantic model for Android permissions and uses this model to locate permissions that can not be matched by descriptions [45]. *CHABADA* [84] utilizes topic model to process descriptions and group apps, and then identifies apps that use abnormal APIs in the same group. In order to uncover the server URLs of mobile apps, Lin et al. [85] use symbolic execution to solve the constraints on the path to get the proper input. Then they utilize dynamic analysis to generate server request message. To identify the vulnerability of brute-force password, given a few legal inputs, Zuo et al. [86] first hook cryptographic APIs to know how the user input is processed. Then they intercept the outgoing messages and infer the semantic of message fields by diffing the messages. In order to understand why some apps are disappeared from the market, Wang et al. [87] compared two snapshots of Google Play apps and they summarized six kinds of apps that are removed.

C. Analysis of Android Apps' Privacy Policies

To analyze the quality of the privacy policies of mobile money apps, Bowers et al. [88] search keywords in these privacy policies and then they manually determine whether the privacy policy comply the guidance provided by US Federal Deposit Insurance Corporation (FDIC) and mobile money industry (GSMA) or not. They also calculated the readability and the language of these privacy policies. The major difference between [88] and *PPChecker* is that [88] manually analyze privacy policies while *PPChecker* combines machine learning and information retrieval technique to analyze them automatically.

Slavin et al. propose a semi-automatic method to find incomplete privacy policy [89]. When detecting incomplete privacy policies, the major technical differences between theirs and *PPChecker* include: 1) they manually select information collection related sentences from privacy policy [89] whereas *PPChecker* accomplishes this task automatically. 2) In terms of code analysis, they only considers APIs [89] whereas *PPChecker* takes into account both APIs and URIs. Moreover, *PPChecker* uses the reachability analysis to avoid infeasible code whereas they do not do it.

Zimmeck et al. propose to combine machine learning and static analysis technique to detect incomplete privacy policy and incorrect privacy policy [90]. Other three kinds of issues detected by *PPChecker* are not considered by their system (i.e., imprecise privacy policy, inconsistent privacy policy, and unfriendly privacy policy). For detecting incomplete and incorrect privacy policy, the techniques used by *PPChecker* are also different from theirs: 1) For privacy policy analysis, before training machine learning classifiers, they use a set of keywords to extract features from privacy corpus whereas *PPChecker* directly trains classifier based on annotated privacy policies. 2) Their static analysis module only examines the APIs of 3 kinds of personal information (i.e., location, device ID, contact) whereas *PPChecker* investigates the APIs and URIs of 14 different kinds of personal information (Section III-C).

Wang et al. [91] also propose to combine privacy policy analysis and static analysis to detect incomplete privacy policy. In detail, their system reports an alert if the sensitive user input is leaked in code but this behavior is not mentioned in privacy policy. However, *PPChecker* regards the privacy policy as incomplete one if the app code accesses/retains personal information by using API/URI and such behavior is not mentioned in privacy policy. Other technical difference in detecting incomplete privacy policy include: 1) Their static analysis module performs static taint analysis on user input while *PPChecker* is from APIs/URIs. 2) Their privacy policy analysis is based on crowd sourcing while *PPChecker* employs machine learning classifier.

In order to determine if the integrated ad lib violate the behavior policy or not, Dong et al. [92] conduct GUI testing to identify the type/location feature of ad views and compare it with the rules summarized from lib behavior policy. The major difference between *PPChecker* and Dong et al. [92] is that: *PPChecker* utilizes static analysis to extract behaviors from app code while they use dynamic analysis. Moreover, *PPChecker* automatically analyzes privacy policies of apps while they manually identify rules from the behavior policy of libs.

Compared with our earlier version [34], this manuscript includes a significant amount of new materials. First, we enhance the capability of *PPChecker*. For the privacy policy analysis, we propose to replace pattern generation algorithm with machine learning algorithm so that seed patterns are not needed (Section III-B). For the static analysis (Section III-C), we enable *PPChecker* to extract the constraint of privacy related behaviors. It will also recover the links of in-app privacy policies. For the description analysis, we involve what's new to infer the permissions used by apps (Section III-D) and detect incomplete/incorrect privacy policy (Section IV-A and Section IV-B). We propose to discover three types of imprecise privacy policies (Section IV-C): 1) the privacy policy does not clearly describes the personal information accessed by the corresponding app; 2) the privacy policy lists the personal information to be accessed, but the app only accesses part of them; 3) the privacy policy mentions that the privacy-related behavior will be conducted under specific condition, but the app does not check whether the condition is satisfied. For detecting the inconsistent privacy polices, we propose to detect the inconsistency between app privacy policy with in-app privacy policy (Section IV-D). We also equip *PPChecker* with the capability of identifying user unfriendly privacy policy (Section IV-E). Second, we perform much more evaluations on *PPChecker* with a larger data set having 2,500 apps. Besides re-conducting the experiments in [34], we add the following new evaluations, including 1) the performance of using machine learning algorithm to analyze privacy policies (Section V-B); 2) the usage of WsN for discovering incomplete/incorrect privacy polices (Section V-C and V-D); 3) the number of imprecise privacy policies (Section V-E); 4) the number of inconsistent privacy policies detected by comparing app privacy policy with in-app privacy policy (Section V-F); 5) user unfriendly privacy policies (Section V-G); and 6) problems in third-party libs' privacy policies (Section V-H).

VIII. Conclusion

To determine whether apps' privacy policies are trustworthy or not, we propose and develop *PPChecker* for automatically identifying five 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 their privacy policies, and found that *PPChecker* can effectively detect questionable privacy policies with high precision. Moreover, the experimental results show that nearly three-quarters of apps have problems (i.e., 74.0%) contain at least one kind of problem. More attention should be paid to the trustworthiness of app privacy policies.

References

[1] A. Varshney, "App stores of future will be based on blockchain, promote transparency," https://goo.gl/fpyC7a, 2017.

[2] TechNode, "China's going to take a bigger chunk of world's $110b app economy in 2018," https://goo.gl/gr4ya6, 2017.

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

[4] 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.

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

[6] 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.

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

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

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

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

[11] "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.

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

[13] 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.

[14] A. D. Rayome, "Google will soon delete apps with no privacy policies from play store," https://goo.gl/peZ3bn, 2017.

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

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

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

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

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

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

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

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

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

[24] C. CHIDGEY, "Google play description v how to write for ranking and conversion," https://goo.gl/fwtYUC, 2015.

[25] A. Store, "Engaging users with app updates," https://goo.gl/H4uQrX, 2017.

[26] "Study report on the privacy policy transparency (internet privacy sweep) of smartphone applications," https://goo.gl/3h0CvU, 2013.

[27] 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.

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

[29] "Four kinds of verbs used by ppchecker," https://drive.google.com/open?id=1kC5Hwgpl8ZZtBgdm9zSEWrZkT75cAgqw, 2018.

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

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

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

[33] Stanford Parser, "Stanford typed dependencies manual," http://nlp.stanford.edu/software/dependencies_manual.pdf, 2016.

[34] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps," in *Proc. DSN*, 2016.

[35] TF-IDF, "Tf-idf: A single page tutorial," http://www.tfidf.com/, 2017.

[36] A. McCallum, D. Freitag, and F. C. Pereira, "Maximum entropy markov models for information extraction and segmentation." in *Icml*, 2000.
[37] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural computation*, 2001.
[38] J. D. Rennie, L. Shih, J. Teevan, and D. R. Karger, "Tackling the poor assumptions of naive bayes text classifiers," in *Proc. ICML*, 2003.
[39] M. Pal, "Random forest classifier for remote sensing classification," *International Journal of Remote Sensing*, 2005.
[40] "Negative vocabulary word list," http://goo.gl/qX7UtK.
[41] E. Grammar, "Adverb clause of condition," https://goo.gl/NWkTkS, 2015.
[42] T. D. Breaux, D. Smullen, and H. Hibshi, "Detecting repurposing and over-collection in multi-party privacy requirements specifications," in *Proc. RE*, 2015.
[43] R. Girju, A. Badulescu, and D. Moldovan, "Learning semantic constraints for the automatic discovery of part-whole relations," in *Proc. NAACL HLT*, 2003.
[44] E. Gabrilovich and S. Markovitch, "Computing semantic relatedness using wikipedia-based explicit semantic analysis." in *Proc. IJCAI*, 2007.
[45] 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.
[46] A. Kachites, "Machine learning for language toolkit," http://mallet.cs.umass.edu/, 2017.
[47] S. Graham, "Getting started with topic modeling and mallet," https://goo.gl/qdVYwD, 2017.
[48] Android, "User interface: Layout," https://goo.gl/xgMySY, 2017.
[49] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: toward discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, 2015.
[50] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. ICSE*, 2017.
[51] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed android applications," in *Proc. ESORICS*, 2015.
[52] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proc. ISSTA*, 2016.
[53] 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.
[54] 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.
[55] 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.
[56] K. Au, Y. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. CCS*, 2012.
[57] 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.
[58] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *ACM SIGPLAN Notices*, 1986.
[59] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proc. CCS*, 2012.
[60] L. Li, J. Klein, Y. Le Traon *et al.*, "An investigation into the use of common libraries in android apps," in *Proc. SANER*, 2016.
[61] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. CCS*. ACM, 2016.
[62] A. De Lucia, "Program slicing: Methods and applications," in *Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001.
[63] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.
[64] M. J. Harrold, G. Rothermel, and A. Orso, "Representation and analysis of software."
[65] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie, "Uiref: analysis of sensitive user inputs in android applications," in *Proc. WiSec*, 2017.
[66] Android, "public class dialog," https://goo.gl/2nhvQk, 2017.
[67] ——, "User interfact: Settings," https://goo.gl/E3Y2rp, 2017.
[68] Regulation (EU) 2016/679, "Art.4 gdpr definitions," https://gdpr-info.eu/art-4-gdpr/.
[69] "Google privacy policy," https://goo.gl/z3C8Xv.
[70] CommonsWare, "System services," https://goo.gl/5VCvRK, 2017.
[71] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android," in *Proc. ASE*, 2015.
[72] Dick Sites, "Compact language detector 2," https://github.com/CLD2Owners/cld2.
[73] R. Flesch, "A new readability yardstick." *Journal of applied psychology*, vol. 32, no. 3, pp. 221–233, 1948.
[74] "How to write plain english," https://goo.gl/STOcgi, 2016.
[75] "Top 90 popular ad libraries," http://goo.gl/GBhXOi, 2015.
[76] "Top 18 popular social libraries," https://goo.gl/9w01mE, 2015.
[77] "The most popular develop tools," https://goo.gl/f2iuGL, 2015.
[78] 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.
[79] A. Yamada, T. H.-J. Kim, and A. Perrig, "Exploiting privacy policy conflicts in online social networks," *CMU-CyLab-12-005*, 2012.
[80] S. Zimmeck and S. M. Bellovin, "Privee: An architecture for automatically analyzing web privacy policies," in *Proc. USENIX Security*, 2014.
[81] F. Schaub, R. Balebako, A. L. Durity, and L. F. Cranor, "A design space for effective privacy notices," in *Proc. SOUPS*, 2015.
[82] 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.
[83] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *Proc. USENIX Security*, 2013.
[84] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. ICSE*, 2014.
[85] C. Zuo and Z. Lin, "Smartgen: Exposing server urls of mobile apps with selective symbolic execution," in *Proc. WWW*, 2017.
[86] C. Zuo, W. Wang, Z. Lin, and R. Wang, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proc. NDSS*, 2016.
[87] H. Wang, H. Li, L. Li, Y. Guo, and G. Xu, "Why are android apps removed from google play? a large-scale empirical study," in *Proc. MSR*, 2018.
[88] J. Bowers, B. Reaves, I. N. Sherman, P. Traynor, and K. Butler, "Regulators, mount up! analysis of privacy policies for mobile money services," in *Proc. USENIX SOUPS*, 2017.
[89] 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.
[90] 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. NDSS*, 2017.
[91] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. Breaux, and J. Niu, "Guileak: Tracing privacy policy claims on user input data for android applications," in *Proc. ICSE*, 2018.
[92] F. Dong, H. Wang, L. Li, Y. Guo, G. Xu, and S. Zhang, "How do mobile apps violate the behavioral policy of advertisement libraries?" in *Proc. HotMobile*, 2018.