The following publication L. Yu, X. Luo, C. Qian, S. Wang and H. K. N. Leung, "Enhancing the Description-to-Behavior Fidelity in Android Apps with Privacy Policy," in IEEE Transactions on Software Engineering, vol. 44, no. 9, pp. 834-854, 1 Sept. 2018 is available at https://doi.org/10.1109/TSE.2017.2730198.

Enhancing the Description-to-Behavior Fidelity in Android Apps with Privacy Policy

Le Yu, Xiapu Luo‡, Chenxiong Qian, Shuai Wang, and Hareton K. N. Leung

***Abstract*—Since more than 96% of mobile malware targets the Android platform, various techniques based on static code analysis or dynamic behavior analysis have been proposed to detect malicious apps. As malware is becoming more complicated and stealthy, recent research proposed a promising detection approach that looks for the inconsistency between an app's permissions and its description. In this paper, we first revisit this approach and reveal that using description and permission will lead to many false positives because descriptions often fail to declare all sensitive operations. Then, we propose exploiting an app's privacy policy and its bytecode to enhance the malware detection based on description and permissions. It is non-trivial to automatically analyze privacy policy and perform the cross-verification among these four kinds of software artifacts including, privacy policy, bytecode, description, and permissions. To address these challenging issues, we first propose a novel data flow model for analyzing privacy policy, and then develop a new system, named TAPVerifier, for carrying out investigation of individual software artifacts and conducting the cross-verification. The experimental results show that TAPVerifier can analyze privacy policy with a high accuracy and recall rate. More importantly, integrating privacy policy and bytecode level information can remove up to 59.4% false alerts of the state-of-the-art systems, such as AutoCog, CHABADA, etc.**

***Index Terms*—mobile applications; privacy policy.**

I. Introduction

The massive success of app economy poses lucrative and profitable targets for attackers. It has been shown that the number of mobile malware has jumped 75% in 2014 [48]. Moreover, while Android has taken up 81.5% market share with millions of apps [43], 96% mobile malware targets Android [48].

Many detection systems based on static analysis [18], [31], [37], [41], [85], [88] and/or dynamic analysis [34], [57], [61], [71], [77], [81] have been proposed to detect mobile malware. However, without well-defined signatures, it is difficult to differentiate between malware and benign apps because they may offer the same functionality. Recent research suggested a promising approach that detects malware by checking its description-to-behavior fidelity (i.e., whether it behaves as advertised [40], [54], [58]). For example, whether a music app collecting users' location information is suspicious or not depends on what it claims to do. These approaches (e.g.,Whyper [54], AutoCog [58]) profile an app's expected behaviors by extracting the semantic meaning from its description, and characterize the app's real behaviors by examining the permissions required by the app. CHABADA [40] identifies abnormal sensitive API usages from apps that are expected to provide similar functionality according to their descriptions.

Although their results are encouraging, there needs a systematic study on assessing an app's description-to-behavior fidelity because of two reasons. First, since an app's description on Google Play has character limit, it cannot detail all behaviors, thus leading to false positives. To remedy this issue, we propose employing an app's privacy policy to obtain more information about its expected behaviors. It is worth noting that more and more apps provide privacy policies for describing their privacy-related behaviors (e.g., 76% of free apps on Google play provide privacy policies for users in 2012 [1]). More importantly, Google recently updated its User Data Policy to require the developers of apps that are in Google play to provide privacy policies. Moreover, since Mar. 15, 2017, Google started removing apps without privacy policies from Google play [64].

Second, since developers may over-claim permissions [35], using permissions to represent an app's behaviors will result in false positives (i.e., "overclaim" security-related behaviors). To address this issue, we propose inspecting an app's bytecode to profile its real behaviors. Therefore, in this paper, we leverage both an app's privacy policy and its bytecode to revisit and enhance its description-to-behavior fidelity by answering the following two research questions:

RQ1 Does an app's privacy policy supply useful information for assessing its description-to-behavior fidelity?

RQ2 Does an app's bytecode provide useful information for measuring its description-to-behavior fidelity?

It is non-trivial to answer these two questions due to the difficulty of analyzing privacy policies and bytecode and correlating them with descriptions and permissions. First, since privacy policies are legal style documents, it is challenging to automatically extract their meanings [89]. We propose a novel data flow model for privacy policy to create semantic patterns, which are then used to identify actions in privacy policy. After that, we use information extraction (IE) and natural language processing (NLP) techniques to identify an app's expected behaviors. Second, since privacy policy, bytecode, description, and permission are different kinds of software artifacts, analysing each of them and then correlating their semantic meanings pose unique challenges. We propose and develop TAPVerifier, a ***T***ext-based ***AP***plication ***V***erification system that integrates the analysis of these four kinds software artifacts together and performs the cross-verification automat-

‡The corresponding author.

Le Yu, Xiapu Luo, Chenxiong Qian, Shuai Wang, and Hareton K. N. Leung are with the Department of Computing in the Hong Kong Polytechnic University.

ically. Overall, our major contributions include:

1) A novel data flow model for privacy policy to define semantic patterns. Based on these patterns, we can automatically identify the collected personal information from apps' privacy policies.
2) We propose and develop TAPVerifier, a novel system for analyzing four kinds of software artifacts, including, privacy policy, bytecode, description, and permission, and conducting cross-verification among them automatically.
3) We conduct extensive experiments to evaluate TAPVerifier. The experimental results show that compared with description, privacy policies are more likely to describe privacy-related behaviours. Furthermore, by using privacy policy and bytecode, TAPVerifier can remove up to 59.4% false alerts generated by the state-of-the-art systems (e.g., Whyper, AutoCog, CHABADA, etc.).

The reminder of this paper is organized as follows. We introduce the background knowledge and the motivating examples in Section II. Section III details how we identify semantic patterns for analyzing privacy policies. Section IV describes the design and the implementation of TAPVerifier. Section V presents the extensive evaluation results and observations. After discussing TAPVerifier's limitations and introducing our future work in Section VI, we present related work in Section VII and conclude the paper in Section VIII.

II. Preliminaries

A. Background

1) Privacy policy: When publishing an app in Google play, developers will provide additional information to help user learn more about the app, such as, description, privacy policy, screenshots, to name a few [8]. An app's description is like an advertisement for promoting the app and attracting more users [73]. To make the description appealing to users, the app's most relevant features are presented. Privacy policy informs users about personal information collection, such as what kind of information will be collected, how information will be used, etc. [14].

2) Android: Each app has an APK file that contains the executable `dex` file, the manifest file (`AndroidManifest.xml`), resource file and other supporting files. Android uses permissions to limit the access to sensitive data or feature on the device. If an app wants to use some features protected by permissions, it must declare corresponding permission in its manifest file.

3) Description-to-Behavior Fidelity: Some researchers propose checking the inconsistencies between an app's description and its real behavior to identify abnormal apps [54] [58] [40]. More precisely, they use the app's description to infer expected behaviors and employ the permissions/APIs used by the app to represent its behaviors.

Whyper [54] and AutoCog [58] combine description and the requested permissions to find improper permissions. After extracting word pairs from description, Whyper and AutoCog map them to permissions. Then, they compare these permissions inferred from the description with the permissions requested by the app. If an app requests some permissions but does not explain it in the description, Whyper/AutoCog will raise an alert. The major difference between these two systems lies in how to construct the semantic model. Whyper builds it by manually analyzing API documents whereas AutoCog creates it by conducting statistical analysis on a large number of descriptions.

CHABADA [40] combines description and the called APIs to find suspicious apps. It first uses LDA [26] to extract topic words from descriptions for grouping apps into different clusters. Then, it identifies suspicious apps with abnormal API usages in the same cluster.

B. Motivating Examples

We use the app "com.tinymission.dailyyogafree" to illustrate how to employ privacy policy to remove false alerts resulted from the insufficiency of description. Fig. 1, Fig. 2, and Fig. 3 show the snippet of the app's code, its privacy policy, and the description, respectively.

```
Location location = ((LocationManager)v0_1).getLastKnownLocation("gps");
if(location == null) {
    location = ((LocationManager)v0_1).getLastKnownLocation("network");
    ......
}
if(location != null) {
    Log.d("", "" + location.getLatitude()
          + " and longitude: " + location.getLongitude());
    ......
}
```

Fig. 1: Snippet of com.tinymission.dailyyogafree's codes. It gets the location information and writes the information to the log.

The types of non-personal data Daily Workout may collect and use include, but are not limited to:

(i) device properties, including, but not limited to unique device identifier or other device identifier ("UDID");

(ii) device software platform and firmware;

(iii) mobile phone carrier;

(iv) geographical data such as zip code, area code and location;

Fig. 2: Snippet of com.tinymission.dailyyogafree's privacy policy.

Your own personal yoga instructor wherever you are! FEATURES:
· Level one 20, 40 and 60 minute workouts · Great for both men and women
· Video demonstrates how to get into each pose · Audio instructions for entire routine
· 30+ poses · 3 predefined routines
>>> Featured in WIRED magazine!!!
Simply Yoga is your own personal yoga instructor. The app contains a 20, 40 and 60 minute yoga routine that step you through each pose. Each pose is demonstrated by a certified personal trainer, so simply choose your workout length and follow along in the comfort of your own home!
If you like Simply Yoga FREE, check out the full version which features:
· A second set of workouts (Level 2) · Create custom routines from all poses
· Landscape mode · Ad-free
>>> Want more workouts? Also check out the "DAILY WORKOUTS" full version app for multiple workouts including ab, arm, butt, cardio, leg and full-body routines. Daily Workouts now also has Pilates, stretch, kettlebell and ball workouts and more!

Fig. 3: Description of com.tinymission.dailyyogafree.

Use privacy policy to explain the necessity of permission. As shown in Fig.1, the app calls the API *getLastKnownLocation()* to get the location information and eventually writes the information into the log. Since invoking this function needs

the permission ACCESS_FINE_LOCATION, the app requires such permission in its manifest file.

Fig.2 lists part of the app's privacy policy, where "Daily Workout" (i.e., "Daily Workout Apps, LLC") is the developer of the app "com.tinymission.dailyyogafree". The item (iv) indicates that the app will collect users' location information and the app requires the permission ACCESS_FINE_LOCATION. In other words, the privacy policy can explain the necessity of requesting this permission.

Use privacy policy to find false alerts. When analyzing this app's description (Fig.3), AutoCog [58] cannot locate any sentences that can explain why the permission ACCESS_FINE_LOCATION is needed, and therefore it raises a permission alert. However, since the privacy policy can explain the necessity, such false alert can be removed by analyzing the privacy policy.

Use code to find false alerts. We use the app "com.ilspl.mahavir" to demonstrate how to use code analysis to remove false alerts due to the overclaimed permissions. The app, "com.ilspl.mahavir", requests permission ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION without explaining it in its description. AutoCog generates an alert. However, our bytecode analysis finds that this app does *not* use any location related APIs. Therefore, such false alert can be removed with the help of bytecode analysis.

III. Semantic Patterns for Privacy Policies

A. Data Flow Model for Privacy Policy

Useful sentence. Not all sentences in privacy policies are relevant to users' personal information. We define "useful sentences" as those sentences that describe what information will be collected by an app. By analyzing useful sentences, we can identify the personal information to be collected by the app. Other sentences are regarded as "useless", and will not be analysed. For example, although the sentence "if you have any question, you can contact us by using the following information" contains sensitive word "contact", we do not analyse it since it talks about how to contact the developer.

Data flow model. Since useful sentences can have diverse formats, we propose a systematic approach to define semantic patterns, and then use such patterns to recognize useful sentences. More precisely, motivated by the categories of privacy elements summarized in [15], [16], we create a data flow model for major components in privacy policies. This model describes how personal information is processed and transmitted and guides us to define semantic patterns.

As shown in Fig. 4, our data flow model has three kinds of actors, including **We**, **You**, and **Third Party**. The former two actors can conduct several actions denoted by their respective blocks. The actor **We** may refer to the app itself, the developer/owner of this app, or the service provider. **We** can collect personal information from the app. The actor **You** refers to the user of an app or service, and **You** can provide information through registering accounts or other channels. The actor **Third Party** (e.g., Ad library) collaborates with **We**, and may receive the information collected by **We**. Our model is general and extensible, and it does not require a privacy policy to include all actions. Moreover, if a new action is identified, we can easily add it to the model.

The model in Fig.4 illustrates how information flows from one actor to another and how it is handled by different actions. We detail each action as follows.

▷ **Data Collection**. This action is usually accompanied with sentences explicitly mentioning which information will be collected by **We**, for example, "*we may collect and process information about your actual location.*".

▷ **Data Storage**. Since **We** may store information in some place after collecting them, the sentences related to this action will reveal the collected information, such as, "*we'll store those contacts on our servers for you to use.*".

▷ **Data Utilization**. Privacy policies also describe what information will be used and the purpose of this behaviour. The sentences about this action will disclose the collected information, such as, "*We may use your location information to display advertisements for businesses*".

▷ **Data Access**. Since some sentences often mention the limited access to the collected personal information, they may provide more details about the personal information, for example, "*Service providers have access to your personal information only to perform services on our behalf.*".

▷ **Data Disclose**. It explains what, when, and how the collected information will be shared with **Third Party** by **We**. Hence, the relevant sentences will give hints to the information, such as, "*We may disclose your information to third parties if we determine that such disclosure is reasonably necessary.*".

▷ **Integrity&Security**. **We** will take some measures to protect the integrity and security of user data. The relevant sentences mention the data collected by **We**, such as "*all sensitive/credit information you supply is encrypted via Secure Socket Layer (SSL) technology.*".

▷ **User Consent**. **You** may accept the privacy policy explicitly by consenting to it. Alternatively, **You** may accept the privacy policy implicitly by using the app. In either case, **We** can acquire the information mentioned in the related sentences, for example, "*Each time you visit the Site or use the Service, you agree and expressly consent to our collection, use and disclosure of the information.*".

▷ **Data Provision**. When **You** register an account, **We** will ask **You** to provide certain information directly. The relevant sentences will present the details, for instance, "*When you register account through website, you will be asked to provide us with your phone number, name and a photo*".

▷ **Data Aggregation**. **We** may combine the information collected from different sources, such as "*we may combine Personal Information with other information, such as combining a precise geographical location with your name.* ".

Service Provision usually briefly introduces the service or data provided to users. Since an app's description provides much more such information than **Service Provision**, we do not analyze it.

B. Semantic Patterns

We define semantic patterns according to the data flow model shown in Fig. 4. More precisely, we first find out the

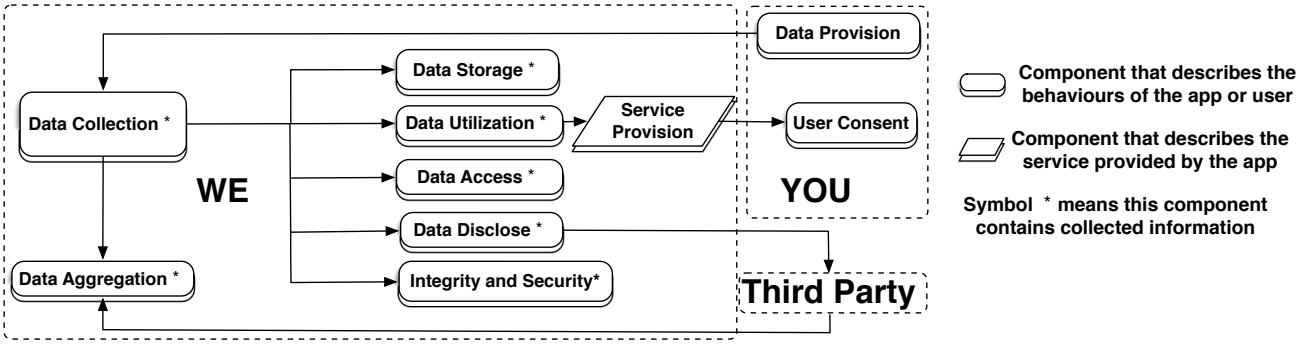


Fig. 4: Data Flow Model for Privacy Policy.

verbs commonly used in different actions, and then define semantic patterns according to those verbs' semantic meanings and common sentence structures in privacy policies.

Verb set. Since the verbs are the basis of semantic patterns, their comprehensiveness would affect the effectiveness of semantic pattern. For example, verb "collect" and "gather" have similar semantic meaning, and may be used interchangeably.

Permission	API Example	Personal Information
WRITE_SETTINGS	*putConfiguration()*	configuration
READ_CONTACTS	*assignContactFromPhone()*	contact
RECORD_AUDIO	*setAudioSource()*	audio source
WRITE_EXTERNAL_STORAGE	*getExternalStorageDirectory()*	external storage directory
WRITE_CONTACTS	*assignContactFromEmail()*	contact
ACCESS_COARSE_LOCATION	*getLastKnownLocation()*	last known location
CAMERA	*setVideoSource()*	video source
RECEIVE_BOOT_COMPLETED	-	-
GET_ACCOUNTS	*getAccounts()*	accounts
READ_CALENDAR	*CalendarContract$Reminders.query()*	reminders
ACCESS_FINE_LOCATION	*addGpsStatusListener()*	Gps status listener

TABLE I: Permissions and their related APIs.

We propose the following approach to automatically construct the verb set that contains verbs with similar meaning. In the first step, since PScout [20] lists the APIs protected by 10 sensitive permissions as shown in Table I, we analyze the method names of these APIs to extract personal information. For example, we extract "last known location" from the API *getLastKnownLocation()*. If the method name only contains a verb (e.g., *Camera.open()*), we collect the personal information from its class name (e.g., "camera").

In the second step, we look for the personal information in a corpus that consists of 500 privacy policies, and locate the corresponding verbs. More precisely, if one sentence contains personal information, we analyze its typed dependency to identify the corresponding verb. For example, since the sentence "we will retain your account information" contains "account", we extract its related verb "retain". We collect the verbs that appear more than 5 times in these privacy policies, and eventually obtain 132 verbs.

In the third step, we divide these verbs into proper categories by comparing them with 23 seed verbs, which are commonly-used verbs in privacy policies as suggested by Anton et al. in [**?**]. We manually group the seed verbs into 12 verb sets according to their semantic meaning, and let them be the seed verbs of each verb set. Then, for the 132 new verbs obtained in the second step, we calculate the semantic similarity between them and the seed verbs in each verb set. Given a new verb, if the similarity between it and the seed verb is higher than a threshold (0.67 by default), we put it into the corresponding verb set.

Finally, 115 new verbs are grouped into 12 verb sets (shown in Table II). Other verbs (e.g. "cancel") are removed since they cannot be classified to any verb set. Table II lists three sample verbs for each verb set. Note that some verbs appear in more than one category. For example, given that "*provide*" is used in a sentence, if the subject is **We**, the sentence belongs to the **Service Provision** action. Otherwise, if the subject is **You**, the sentence should belong to the **Data Provision** action of **You**.

#	Verb Set(Action Name)	Example verbs
1	$VP_{collect}$ (**Data Collection**)	collect, gather, capture,...
2	$VP_{contain}$ (**Data Collection**)	contain, include, involve,...
3	VP_{access}(**Data Access**)	access, read, see,...
4	$VP_{access-control}$ (**Data Access**)	limit, restrict, gain,...
5	VP_{store} (**Data Storage**)	store, reserve, log,...
6	VP_{use} (**Data Utilization**)	use, process, link,...
7	$VP_{disclose}$ (**Data Disclose**)	share, sell, disclose,...
8	VP_{allow} (**All actions**)	allow, disallow, permit,...
9	$VP_{provide}$ (**All actions**)	provide, supply, offer,...
10	$VP_{consent}$ (**User Consent**)	consent, agree, assent,...
11	$VP_{protect}$ (**Integrity&Security**)	protect, encrypt, decrypt, ...
12	$VP_{combine}$ (**Data Aggregation**)	aggregate, combine, merge, ...

TABLE II: Common verbs and their related actions.

The following paragraphs will detail the semantic patterns for each action. We determine these semantic patterns by first looking for the main verbs (e.g., those shown in Table II) in real privacy policies and then manually reading the corresponding sentences and extracting the patterns. To ease the presentation of semantic patterns, *resource* represents the place where the collected information will appear. We use VP_{*}^{pass} to indicate the corresponding *passive* voice of the verbs.

Data Collection. Its semantic patterns include:
Pattern_DC 1: $sbj\ VP_{collect}\ resource$
Pattern_DC 2: $resource\ VP_{collect}^{pass}$
Pattern_DC 3: $sbj\ VP_{collect}\ VP_{contain}\ resource$
Pattern_DC 4: $sbj\ VP_{allow}\ obj\ to\ VP_{collect}\ resource$
Pattern_DC 5: $sbj\ VP_{allow}^{pass}\ to\ VP_{collect}\ resource$

The sentences related to this action will describe the collected information directly, such as "*We will collect any information contained in such communication*" (Pattern_DC 1) or "*Personal data about you will be collected*" (Pattern_DC 2). They may ask for the permission to collect some information, for example, "*Cookies allow us to collect technical*

and navigational information" (Pattern_DC 4) or "*We are allowed to collect and store the following personal information*" (Pattern_DC 5). Pattern_DC 3 indicates a special class of descriptive sentences that enumerate individual collected information, for instance, "*Examples of the information we collect include name, mobile phone number*". The subject and the object form the part-whole relation [39], where name and mobile phone number are part of the collected information.

Data Storage. Its semantic patterns include:
Pattern_DS 1: $sbj\ VP_{store}\ resource$
Pattern_DS 2: $resource\ VP^{pass}_{store}$
Pattern_DS 3: $sbj\ VP_{allow}\ obj\ to\ VP_{store}\ resource$
Pattern_DS 4: $sbj\ VP^{pass}_{allow}\ to\ VP_{store}\ resource$

Pattern_DS 1-4 are similar to Pattern_DC 1,2,4,5 in **Data Collection**, but the verb set $VP_{collect}$ is replaced with VP_{store}. Example sentences include: "*We will store and use your e-mail address*" (Pattern_DS 1), "*Your e-mail address will be stored by us*" (Pattern_DS 2), "*Persistent cookies also allow us to store your preferences*" (Pattern_DS 3), and "*We are allowed to store your contact information*" (Pattern_DS 4).

Data Utilization. Its common semantic patterns include:
Pattern_DU 1: $sbj\ VP_{use}\ resource$
Pattern_DU 2: $resource\ VP^{pass}_{use}$
Pattern_DU 3: $sbj\ VP_{allow}\ obj\ to\ VP_{use}\ resource$
Pattern_DU 4: $sbj\ VP^{pass}_{allow}\ to\ VP_{use}\ resource$

Pattern_DU 1,2 are similar to Pattern_DA 1,2, but they use the verbs in VP_{use}. For instance, "*We will utilize cookies for identifying your language settings of your device*" (Pattern_DU 1) and "*The personal data collected will be used for handling such enquiry*" (Pattern_DU 2). Pattern_DU 3, 4 allow "us" to use personal information. An example sentence is "*You allow us to process your personal data*" (Pattern_DU 3) and "*We are allowed to use that information*" (Pattern_DU 4).

Data Access. Its common semantic patterns include:
Pattern_DA 1: $sbj\ VP_{access}\ resource$
Pattern_DA 2: $resource\ VP^{pass}_{access}$
Pattern_DA 3: $sbj\ VP_{allow}\ obj\ to\ VP_{access}\ resource$
Pattern_DA 4: $sbj\ VP^{pass}_{allow}\ to\ VP_{access}\ resource$
Pattern_DA 5: $resource\ ADJ_{access}\ to\ sb$
Pattern_DA 6: $sbj\ "keep\ ability\ "to\ VP_{access}\ resource$
Pattern_DA 7: $sbj\ VP_{accesscontrol}\ "access\ to\ "resource\ to\ sb$

Pattern_DA 1-4 are the same as Pattern_DC 1,2,4,5 in **Data Collection**, but the verb set $VP_{collect}$ is replaced with VP_{access}. Pattern_DA 5 uses adjective to indicate that the information can be collected, for example, "*Your information is accessible to us*". Pattern_DA 6 explains the app's ability to collect information, such as, "*We keep ability to access your personal information*". Pattern_DA 7 denotes limited access to certain information, for instance, "*We limit access to your personal information to those employees*".

Data Disclose. Its common semantic patterns include:
Pattern_DD 1: $sbj\ VP_{disclose}\ resource$
Pattern_DD 2: $resource\ VP^{pass}_{disclose}$
Pattern_DD 3: $sbj\ VP_{allow}\ obj\ to\ VP_{disclose}\ resource$
Pattern_DD 4: $sbj\ VP^{pass}_{allow}\ to\ VP_{disclose}\ resource$

For data disclose action, we define four semantic patterns, just like patterns defined in **Data Utilization**, but VP_{use} is replaced with $VP_{disclose}$. Example sentences include: "*We will transfer your individual information to third parties when necessary*"(Pattern_DD 1), "*Your personal information will be disclosed to such third parties*" (Pattern_DD 2), "*You allow us to share your personal information with another company*" (Pattern_DD 3), and "*We are allowed to sell your data to others*" (Pattern_DD 4)

Integrity&Security. Its common semantic patterns include:
Pattern_IS 1: $sbj\ VP_{use}\ resource\ to\ VP_{protect}\ resource$
Pattern_IS 2: $resource\ VP^{pass}_{protect}$

Pattern_IS 1 is in active voice, for example, "*We will use appropriate security safeguards to protect your personal information against loss, theft, and unauthorized access*". Pattern_IS 2 is in passive voice, such as, "*The sensitive information will be encrypted*".

User Consent. It has the following pattern meaning that the user consents to the information collection:
Pattern_UC: $sbj_{you}\ VP_{consent}\ to\ something$

This pattern matches sentences like "*you are consent to the collection of your personal information*". Since the **User Consent** action describes the behaviors of the user, TAPVerifier will not match or analyze relevant sentences.

Data Provision. It has the following pattern indicating that the user will provide certain information to the app.
Pattern_DP: $sbj_{you}\ VP_{provide} resource$

This pattern represents sentences like "*You should provide a telephone number and an email address*".

Data Aggregation. It usually uses the following pattern:
Pattern_DAG: $sbj\ VP_{combine}\ resource\ with\ resource$

Pattern_DAG describes that resources will be combined together, such as "*We will combine the information you provide with information from other visitors*".

After defining the common sentence structures for different actions, we combine these sentence structures and get nine general semantic patterns (listed in Table III). We use VP_* to represent the verb sets (1,3,5,6,7,9,11) in Table II. Thus, we can use one general semantic pattern to represent multiple sentence structures.

IV. TAPVERIFIER

A. Architecture

Fig.5 shows the architecture of TAPVerifier, which takes in an app's privacy policy, description, and APK file. The *privacy policy analysis module* (Section IV-B) processes the privacy policy and outputs a list of information that will or will not be collected. Since many apps contain third-party libraries that have separate privacy polices, TAPVerifier will also process the third-party libraries' privacy policies.

The *permission and code analysis module* (Section IV-C) analyzes the manifest file and the dex file to construct an App Property Graph (APG) for representing the app [56]. APGs are stored in a graph database. Then the module will look for sensitive APIs/URIs and the corresponding callers.

Since we focus on privacy policies, TAPVerifier reuses the start-of-the-art systems (i.e., AutoCog and Whyper) to analyze descriptions (Section IV-D). The output contains permission alerts from these systems.

#	Semantic Pattern	Sample Sentences
1	$sbj\ VP_*\ resource$	We would collect your location information.
2	$resource\ VP_*^{pass}$	Your location would be collected.
3	$sbj\ VP_*\ VP_{contain}\ resource$	The information we collect include: name, age, birthday.
4	$sbj\ VP_{allow}\ obj\ to\ VP_*\ resource$	You allow us to access your personal information.
5	$sbj\ VP_{allow}^{pass}\ to\ VP_*\ resource$	We are allowed to access your personal information.
6	$resource\ ADJ_{access}\ to\ sb$	Your location information is accessible to us.
7	sbj "$keep\ ability$ "$to\ VP_{access}\ resource$	We keep the ability to access your location information.
8	$sbj\ VP_{access-control}$ "$access\ to$ "$resource\ to\ sb$	We limit access to your personal data stored in our server to employee.
9	$sbj\ VP_{combine}\ resource\ with\ resource$	We will combine your geographical location with your name.

TABLE III: General semantic patterns.

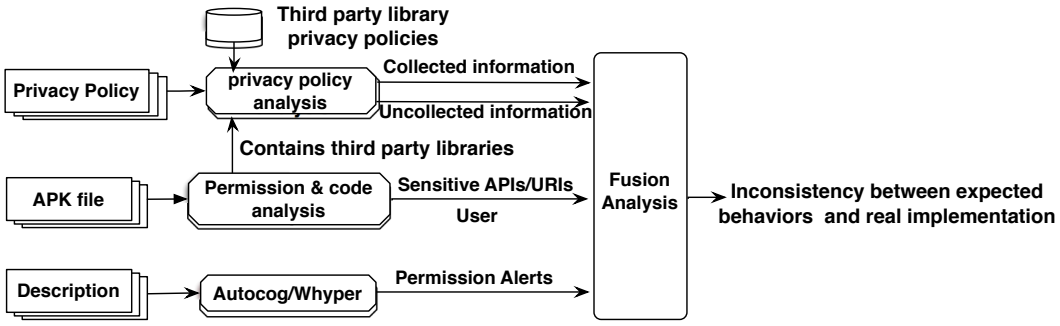


Fig. 5: TAPVerifier's Architecture

The *fusion analysis module* (Section IV-E) leverages the expected behaviors extracted from privacy policy and description to detect the inconsistency between expected behaviors and real behaviors reflected from permissions and bytecode. It also inspects the code to remove the false alerts due to overclaimed permissions.

B. Privacy Policy Analysis

1) Overview: We employ IR and NLP techniques to process privacy policies. As shown in Fig. 6, the procedure has the following major steps. The pre-processing step (Section IV-B2) extracts the content from privacy policy files in HTML format and splits it into distinct sentences. The syntactic parsing step (Section IV-B3) parses distinct sentences and generates syntactic trees and typed dependencies.

The pattern matching step (Section IV-B4) identifies useful sentences by matching sentences with semantic patterns. The collected information extraction step (Section IV-B5) identifies the collected information from useful sentences. The negation analysis step (Section IV-B6) determines negative sentences due to negation words. Finally, the privacy policy analysis module outputs the collected (uncollected) information.

2) Pre-processing: Since the privacy policy file is saved in HTML format, we use Beautiful Soup [10] to extract the content. For the ease of processing, we only keep English letters and some specified punctuation symbols (such as comma, period quotation marks, colon, etc), and remove all non-ascii symbols and some meaningless ascii symbols (such as "*", "#", "$", etc.).

After that, we use the natural language toolkit (NLTK) [7] to split the text into sentences, because it has a pre-trained Punkt tokenizer for English and contains a model for abbreviation words, collocations, and words that start sentences.

3) Syntactic Parsing: For each sentence, TAPVerifier employs the Stanford Parser [30] to analyze it and generate the sentence's syntactic tree and its words' dependency relations. Such data serves as the basis for pattern matching and collected information extraction. For example, Fig.7 shows the result of parsing the sentence: "*we would use your location, account information when you use our app.*", which includes a prase tree and the typed dependencies.

The parse tree starts from S, which denotes the start of a sentence or a clause. The Stanford Parser divides the sentence into phrases, each of which occupies one line in the hierarchy structure. The parser also attaches *part-of-speech* (POS) tags to words and phrases according to their syntax behaviors. Common POS tags for English include noun, verb, adjective, adverb, pronoun, etc. In Fig.7, NP means noun phrase, VP denotes verb phrase, PRP indicates pronoun, VB represents verb, and NN expresses noun. The typed dependencies shows the relation information between words in multiple lines. Each line starts with the relation name, followed by the governor word and the dependent word. Common relations include $nsubj$ that means the subject, $dobj$ that represents the direct object, and $root$ that points to the root word of the sentence.

4) Pattern Matching: Pattern matching is the core component of our privacy policy analysis module. It identifies all useful sentences and their corresponding semantic patterns based on the syntactic information extracted from the syntactic parsing step. Those sentences that cannot be mapped to any semantic patterns will be removed. The pattern matching algorithm is shown in Algorithm 1, where a sentence is a useful one if it matches any one of the 9 general semantic patterns defined in Table III. The useful sentences found in this step and their corresponding semantic patterns will form the input of the collected information extraction step.

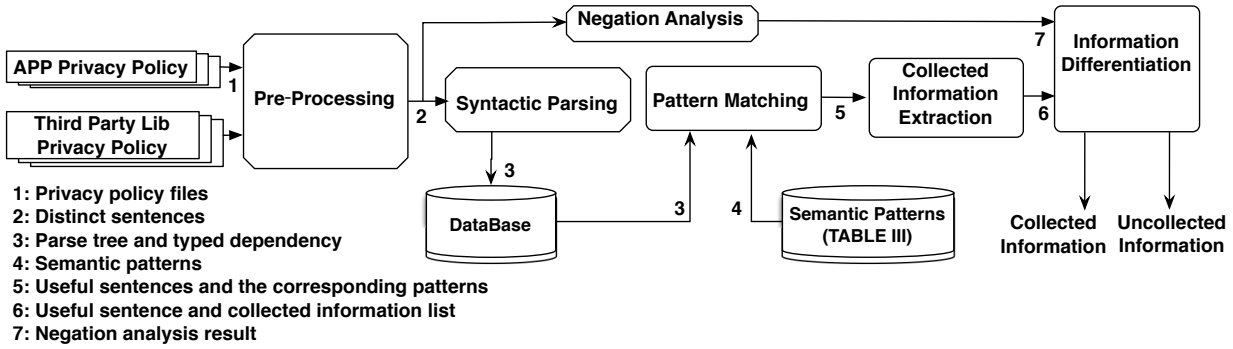


Fig. 6: The procedure of privacy policy analysis

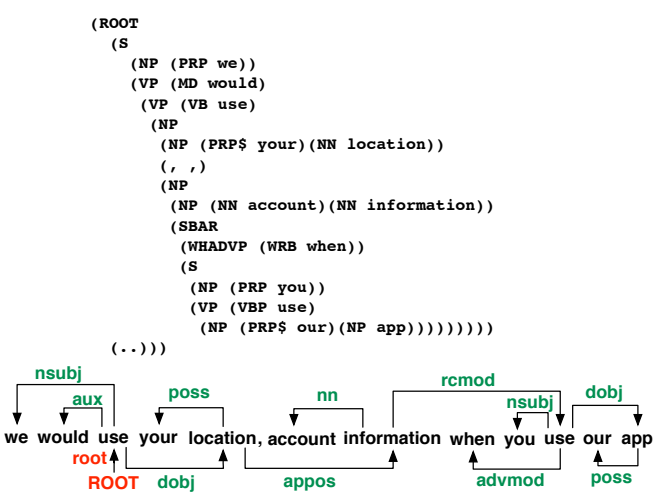


Fig. 7: Parse tree and typed dependencies of the sentence "*we would use your location, account information when you use our app.*"

The function $getWord(query\_relation, query\_word)$ returns a set of words that have the relation $query\_relation$ with the word $query\_word$ in the typed dependencies. The function $getVerbCate(query\_verb)$ is used to get the verb set, which $query\_verb$ belongs to. For example, "*collect*" belongs to $VP_{collect}$. The output of the function $len(query\_set)$ is the number of words in $query\_set$. The function $getWordAfter(str, keyword)$ searches for the sentence str, and returns the first word after $keyword$.

Here, we just use pattern 1 and 2 as examples to explain this algorithm. Most actions in the data flow model contain semantic patterns in active voice (like general semantic pattern 1 in Table III) and passive voice (like general semantic pattern 2 in Table III). To match the general semantic pattern 1 and 2, we look up the dependency relationship in order to find the word that has a "root" dependency relation with the node $Root$. In the following section, we call this word $root\_word$. Since sample sentences 1 and 2 in Table III use "collect" as $root\_word$, they will be matched in this step.

The category of the $root\_word$ affects the action of the corresponding sentence. For instance, verb "*collect*" indicates that this sentence belongs to **Data Collection**, but "*use*" indicates that this sentence belongs to **Data Utilization**. Therefore, after getting $root\_word$, in line 2, we look up Table II to find its corresponding verb set and determine the sentence's action.

Input: str_sent : sentence to match; $Dep\_Relations$: Typed Dependency Relation list. $Root$ is the dummy word that governs the root word in the typed dependency.
Output: 1,2,3,..,8,9: General semantic pattern number; 0: Match fail.

1 $root\_word = getWord(\text{"}root\text{"}, Root)$
2 $cate = getVerbCate(root\_word)$
3 **if** $cate == VP_*$ **then**
4 | // try to match pattern 1,2
5 | **if** $len(getWord(\text{"}auxpass\text{"}, root\_word)) == 0$ **then**
6 | | return 1;
7 | **end**
8 | return 2;
9 **else if** $cate == VP_{contain}$ **then**
10 | // try to match pattern 3
11 | **for** sbj *in* $getWord(\text{"}nsubj\text{"}, root\_word)$ **do**
12 | | **for** mod_word *in* $getWords(\text{"}rcmod\text{"}, sbj)$ **do**
13 | | | **if** $getVerbCate(mod\_word) == VP_{collect}$ **then**
14 | | | | return 3;
15 | | | **end**
16 | | **end**
17 | **end**
18 **else if** $cate == VP_{allow}$ **then**
19 | // try to match pattern 4,5
20 | $passive\_words = getWord(\text{"}auxpass\text{"}, root\_word)$
21 | **for** $verb \in getWord(\text{"}xcomp\text{"}, root\_word)$ **do**
22 | | **if** $getVerbCate(verb) == VP_*$ **then**
23 | | | **if** $len(passive\_words) == 0$ **then**
24 | | | | return 4;
25 | | | **end**
26 | | | return 5;
27 | | **end**
28 | **end**
29 **else if** $cate == ADJ_{access}$ **then**
30 | // try to match pattern 6
31 | return 6;
32 **else if** *"able to" in str_sent* ||*"keep ability to" in str_sent* **then**
33 | // try to match pattern 7
34 | **if** *"able to" in str_sent* **then**
35 | | $verb = getWordAfter(str\_sent, \text{"}able\ to\text{"})$
36 | **else**
37 | | $verb = getWordAfter(str\_sent, \text{"}keep\ ability\ to\text{"})$
38 | **end**
39 | **if** $getVerbCate(verb) == VP_{access}$ **then**
40 | | return 7;
41 | **end**
42 **else if** $cate == VP_{access-control}$ *&&"access to" in str_sent* **then**
43 | // try to match pattern 8
44 | return 8;
45 **else if** $cate == VP_{combine}$ **then**
46 | // try to match pattern 9
47 | return 9;
48 **else**
49 | return 0; // all pattern match fail, return 0;
50 **end**

Algorithm 1: Semantic Pattern Match.

We use different methods to extract the collected information from active voice sentences and passive voice sentences. For an active voice sentence (e.g., Table III sample sentence 1), the collected information is the object of $root\_word$, while

in a passive voice sentence (e.g., Table III sample sentence 2), the collected information is the subject of $root\_word$. After successfully matching $root\_word$, we check whether a sentence uses passive voice in line 5 in order to determine which general semantic patterns (i.e., 1 or 2) this sentence belongs to. This is achieved by counting the number of words that have "auxpass" dependency relation with $root\_word$ in the dependency relation list. Note that "auxpass" means "passive auxiliary".

After identifying the semantic pattern according to the $root\_word$, we check the action executor. If the semantic pattern belongs to **We**, the action executor should not be **You**. For example, if the $root\_word \in VP_{provide}$, this sentence's action executor should not be **We**, because only the personal information provided by users will be considered.

5) Collected Information Extraction: For each useful sentence, TAPVerifier locates the collected information according to the semantic pattern that matches the sentence. In other words, once a general semantic pattern is determined, TAPVerifier looks for the corresponding *resource* as shown in Table III in the parse tree. Note that we do not extract the noun phrases in the conditional clauses. For example, given the sentence "*we would use your location information when you visit our website.*", we will extract the noun phrase "your location information", and ignore the noun phrase "our website". Moreover, we remove stop words for improving the accuracy. For example, the noun phrase "your location information" becomes "location information".

To improve the performance, we adopt ARKref [53] to conduct the co-reference resolution. ARKref is a rule-based coreference system that uses Stanford Parser [30] to analyze sentences and marks most NP as mentions. Then, it uses a set of patterns to find potential antecedents for each mention. For pronominal mention, it selects the antecedent candidate with the shortest syntactic path distance as the entity to which the pronoun points. If one pronoun denotes the collected information, the corresponding noun will be added to the list of collected information. Besides the personal information, we also record the corresponding verb and write it to the output file. The verb can be used to improve the accuracy of the fusion analysis module as explained in Section IV-E.

6) Negation Analysis: When performing the negation analysis, we consider the following negative words, including negative determiners (e.g., "*no*", "*neither*"), negative adjectives (e.g., "*unable*", "*improper*"), negative nouns (e.g., "*nobody*", "*none*"), verbs (e.g., "*prevent*", "*prohibit*", "*forbid*") with negative connotation, adverbs (e.g., "*hardly*", "*scarcely*", "*barely*") [79], and coordinating conjunctions that present a contrast or exception.

To analyze complex sentences with negative words, we consider not only the common cases with a single negative word, but also the scenarios with more than one negative words (e.g., double negation sentences). For example, in the sentence "*we will not collect any personal information but account name*", "but" is a conjunction and it introduces a phrase contrasting with what has been mentioned. Thus, the negation analysis result of "account name" becomes the opposite of the result of "personal information".

More precisely, in the first step, we check the subject and main verb related words. If these words contain negative words, we regard the sentence as a negative one. Otherwise, it is a positive one. For a negative sentence, we first label all personal information as not being collected. For example, for the sentence "*we will not collect any personal information but account name*", we first label both "personal information" and "account name" as not being collected.

After that, if the sentence contains conjunctions that present a contrast or exception (e.g., "but"), we locate the information in the clauses or sentences following the conjunction, and negate its current result (i.e., from "not being collected" to "being collected" or from "being collected" to "not being collected"). For the sentence "*we will not collect any personal information but account name*", since "account name" is in the clause following "but", we change its result from "not being collected" to "being collected". If the clauses/sentences following the conjunctions also contain negative words or conjunctions that present a contrast or exception, we will repeat this analysis on them.

According to our experiences in manually analyzing apps' privacy policies, we find that most negative sentences in privacy policy have only one negative word and a small number of sentences belong to the double negation cases. Very few sentences use more complex structure. It may be due to the fact that most guidelines (e.g., those from Google or privacy commissioners around the world) suggest using simple language to clearly present the content so that users can easily understand the privacy policy. Since it is time-consuming to manually check all sentences, we randomly select 500 negative sentences from 200 privacy policies as samples and manually read them. By focusing on compound sentences, complex sentences, and compound-complex sentences, we found 11 sentences with complex structures (account for 2.2%), which cannot be successfully processed by the current version of TAPVerifier. We summarize the structures of these sentences as follows and propose potential solutions for handling them in future work.

Structure 1: 3 sentences utilize negation words to modify the nouns and have positive and negative meanings at the same time. For example, in the sentence "we may share generic aggregated demographic information not linked to any personal identification information", the demographic information will be shared but it will not be linked to personal identification information. TAPVerifier identifies the root word "share" and hence regards the sentence as a positive sentence because the root word is not modified by any negation words.

Structure 2: 6 sentences are compound sentences, part of which includes a negative sentence. For example, "Anonymous Data is collected or generated and is not associated or linked to Personal Data". In this sentence, "Anonymous Data" is collected/generated but it is "not" associated/linked to personal data. Since TAPVerifer cannot identify the positive and negative meanings at the same time, it only locates the root word "collected" and regards the sentence as a positive one.

Structure 3: 2 sentences have the pattern "may and may not ....". TAPVerifier cannot determine whether the behavior

will be conducted or not. For example, consider the sentence "you understand and agree that Tap Slots may or may not prescreen content".

To handle these sentences, we will perform fine-grained analysis on them in future work. For example, for the sentences of **Structure 1**, TAPVerifier should identify that "generic aggregated demographic information" is the object of the verb "share" and it is also modified by the phrase "not linked to any personal identification information". Note that the negation analysis result of the verb "share" is positive whereas the result of the phrase "not linked to any personal identification information" is negative. We should combine them together to draw the conclusion. For the sentences of **Structure 2**, each distinct sentence contained in the compound sentence should be extracted and analyzed respectively, and then the final conclusion should take into account the result of each distinct sentence. For example, the negation analysis result of the sub-sentence "Anonymous Data is collected or generated" is positive whereas the result of the sub-sentence "is not associated or linked to Personal Data" is negative. For the sentences of **Structure 3**, two different negation analysis results should be generated at the same time (i.e., positive sentence for "may" and negative sentence for "may not") and then seek the decision from users.

7) Privacy Policies of Third-Party Libraries: Since many apps contain third-party libraries that have their own privacy policies, given an app with third-party libraries, TAPVerifier will also analyze their privacy policies individually. To prepare the database for popular third-party libraries' privacy policies, we download the SDKs and the privacy policies of top 83 Ad libraries listed in [17], 9 social libraries [5], and 24 most commonly used development tools [9]. After filtering out the privacy policies written in languages other than English, we use TAPVerifier to analyze the privacy policies of 46 Ad libraries, 9 social libraries, and 24 development tools.

C. Code and permission analysis

TAPVerifier improves our static analysis framework, VulHunter [56], and employs the enhanced version to analyze each app *without* source code.

1) Static analysis module: Given an APK file, TAPVerifier extracts the `AndroidManifest.xml` and the `dex` file. If the app is hardened, we leverage the unpacking tool DexHunter [86] to recover the `dex` file. By parsing the `AndroidManifest.xml` file, TAPVerifier finds out all components and the required permissions. Then, we use Soot [74] with Dexpler [23] to transform the Dalvik code in `dex` file to the intermediate representation *Shimple*. *Shimple* is very similar to *Jimple*, another intermediate representation, with the additional support of static single assignment(i.e., SSA) [11]. SSA can simplify the data flow analysis because it guarantees that each variable is assigned only once and is defined before it is used. Based on the class hierarchy (CHA [33]) and the intermediate representation, we create an Android property graph (APG) [56] that integrates abstract syntax tree (AST), interprocedure control-flow graph (ICFG), method call graph (MCG), and system dependency graph (SDG) of the app.

When building MCG, due to Android's event-driven nature, we carefully handle the callbacks used by the framework. To avoid missing the callbacks of event listeners, we add a connection between the invocation (e.g., *setOnClickListener()*) to the corresponding object's callback (e.g., *onClick()*). Moreover, if the developer extends thread related class (e.g., *java.lang.Thread*), we add a connection between the *start()* method and the *run()* method. EdgeMiner [29] summarizes the implicit control flow transitions through the Android framework. To improve the precision of our static analysis system, we leverage the transitions found by EdgeMiner to enhance our MCG.

The life-cycle methods of components are called by framework to start, pause, resume, or shut down the app. For example, when loading one *Activity*, three life-cycle methods (i.e., *Activity.onCreate()*, *Activity.onStart()*, *Activity.onResume()*) are called sequentially. To model such transitions between life-cycle methods, FlowDroid [19] create a dummy main method for each component. By referring to the control flow graph of the dummy main method proposed by FlowDroid, we create the connections between life-cycle methods (e.g., from *Activity.onCreate()* to *Activity.onStart()*, from *Activity.onStart()* to *Activity.onResume()*).

We define source functions as the APIs through which an app can collect information from device. For example, *getDeviceId* can be used to get the device ID [60]. Apart from APIs, app can also gain information by querying the content provider with URIs. For example, by calling *ContentResolver.Query()* and using URI "content://com.android.calendar" as parameter, the app can read the user's calendar. We define sink functions as the APIs that can transmit information through internet, SMS, file, log, or other channels [60].

The Inter-Component Communication (ICC) model of Android enables the components to exchange data through Intent. To handle the inter-component communication, we use `IccTA` [45] to map a component's launch functions to the corresponding callbacks. FlowDroid [19] is the state-of-art static taint analysis system. The source to sink paths found by FlowDroid are also included when building SDG.

Since the developer can use Java reflection to invoke APIs, we utilize DroidRA [46] to reveal the APIs invoked via reflection and then update APG if need.

2) Traversals: After building graphs for each app, we perform traversal to find the APIs and content providers protected by permission. To find the information obtained by APIs, we check all `invoke_stmt` and `assign_stmt` statements. If source functions are called, we infer that the corresponding information is used by the app.

To find the information obtained by content provider, we use data dependency relation as the directed edge and do depth-first search from the URI parameter of the content provider query functions(*e.g.*, *ContentResolver.Query()*). All possible URI strings and URI fields appear on the search paths are recorded. If sensitive URI strings(or URI fields) appear on the path, we expect that the corresponding information is used.

3) Permission analysis: Certain permissions are required when an app calls sensitive APIs or queries content providers with some URIs. To find the permissions that an app actually

requires to call APIs and use content providers, we employ the mapping between the APIs (URI strings and URI fields) and the permissions provided by PScout [20]. Similar to **(1)**, we search all called APIs, used URI strings and fields of the app. If a specified API or URI is used, we conclude that the corresponding permission is required by the app.

Some permissions are used by third-party libraries. In order to find such permissions, we maintain a white list that contains class name prefixes of commonly used third-party libraries. After finding the API, URI string, or URI fields that require sensitive permissions, we check the class name. If the class name has the same prefix as a library, we expect that the corresponding permission is used by the third-party library.

If the developers use the obfuscation techniques (e.g., ProGuard) to hide the class name, method name, and variable name, the third-party libraries may not be correctly identified. To address this issue, we propose a two-stage approach by first leveraging the available de-obfuscation tool [25] to recover the names and then applying TAPVerifier. More precisely, we employ DeGuard proposed by Bichsel el at. [25], which uses non-obfuscated Android apps to learn probabilistic graphical models for recovering the class name, method name, and variable name.

D. Description Analysis

Since AutoCog handles more permissions with better performance than Whyper [58], we use it to process descriptions. AutoCog maps the sentences of a description to permissions. Its description-to-permission relatedness (DPR) module provides a list of governor-dependent pairs for each permission. For example, governor-dependent pair <"update", "location"> can be mapped to `ACCESS_FINE_LOCATION` permission.

Given a description, TAPVerifier obtains the text content from the HTML file and then splits the text into distinct sentences. After using the Stanford parser to parse each sentence, TAPVerifier extracts all possible governor-dependent pairs from the sentence and compares them with the pairs provided by AutoCog's DPR module. If the comparison result exceeds the threshold (i.e., 0.67 in [58]), this sentence is mapped to the corresponding permission. After processing all sentences and all permissions, if any permission cannot be mapped to any sentences in the description, AutoCog raises an alert [58].

E. Fusion Analysis

Since the privacy policy provides information about an app's expected behaviors, we first describe how to use it to explain the necessity of permission (Section IV-E1).

Existing systems (e.g., AutoCog) identify the inconsistency between an app's description and its requested permissions. If the app requests sensitive permissions without mentioning them in its description, AutoCog will raise an alert. The fusion analysis module further improves the performance of AutoCog from two aspects. First, we conduct the privacy policy analysis to remove the false alerts of AutoCog (Section IV-E2). Second, we perform the bytecode analysis to remove its false alerts(Section IV-E3).

We describe the methods in this section, and present the experimental results and insights in Section V.

1) Use privacy policy to explain the necessity of permission: When mapping the privacy policy to different permissions, different kinds of permissions are processed separately. AutoCog [58] considered 11 different kinds of permissions. Since the permission `RECEIVE_BOOT_COMPLETED` cannot be mapped to any personal information, we map privacy policy to the remaining 10 permissions. We divide the 10 permissions into two categories and describe their mapping methods respectively.

The first category of permissions allows the app to collect personal information, including `READ_CONTACTS`, `RECORD_AUDIO`, `ACCESS_COARSE_LOCATION`, `CAMERA`, `GET_ACCOUNTS`, `READ_CALENDAR`, and `ACCESS_FINE_LOCATION`. To map privacy policy to these permissions, we correlate the collected information in privacy policies with the resources protected by permissions. More precisely, for each permission, we get the APIs under its protection using PScout [20] and define the corresponding resources by analyzing the permission's description and the APIs' document. For example, the permission `RECORD_AUDIO` is mapped to resources like *audio*, *microphone*, *speech*, etc. This step is similar to building the semantic graph in Whyper but we do not need to enumerate the corresponding verbs. Then, we calculate the similarity of a pair of the collected information from privacy policies and the resource from permissions using ESA [38], which is a WiKi-based semantic analysis system. If the result exceeds the threshold, the collected information (or the sentence in privacy policies) can be mapped to the resource (or the permission). We currently set the threshold to be 0.67.

The second category of permissions stores personal information collected by the app, including `WRITE_SETTINGS`, `WRITE_EXTERNAL_STORAGE`, `WRITE_CONTACTS`. For these permissions, we take into account the verb of personal information when mapping personal information to them. For example, if an app requests `WRITE_CONTACTS` permission and its privacy policy says "we will read your contact", we still cannot map this sentence to the permission, because the verb "read" is weaker than "write". However, if the verb changes to others like "modify", "change", or VP_{store}, then we can map the sentence to `WRITE_CONTACTS`.

2) Use privacy policy to remove false alerts of description analysis module: We can either use apps' or third-party libraries' privacy policies to remove false alerts.

- Using apps' privacy policies. When developers request some permissions and mention such behaviors in an app's privacy policy instead of its description, we can leverage the app privacy policy to remove false alerts resulted from the description analysis. More precisely, after getting the alerts generated by the description analysis module (i.e., `AutoCog`), TAPVerifier locates the suspicious permissions that can be explained by the app privacy policy and then removes them, thus improving the accuracy of description analysis module.

- Using third-party libraries' privacy policies. When the integrated third-party library requests some permissions and mention such behaviors in the library's privacy policy instead of the app description, we can leverage the library's privacy policy to remove false alerts generated from the description analysis. More precisely, after getting alerts generated by the description analysis module (i.e.,`AutoCog`), TAPVerifier check the user of the permission. If the permission is used by some third-party library and the library's privacy policy explains the use of such permission, the suspicious permission can be removed.

3) Use bytecode level information to remove false alerts of description analysis module: Since apps may claim more permissions than they need [35], we cannot map their descriptions and/or privacy policies to some permissions. To remove such false positives, given a permission and an app, we perform traversals on the app's method call graph and system dependency graph to check whether it uses APIs or accesses content providers protected by the permission. If the query returns null, the permission is over-claimed by the app and related alerts will be removed.

V. Experiments and Evaluation

We have implemented TAPVerifier in 6,310 lines of python code and the 1,510 lines java code on top of our system *VulHunter* [56] We have also developed a crawler (1,334 line of python codes) to automatically fetch apps' APK files, descriptions, and privacy policies from Goolge play [3].

We conduct experiments to answer the following Q1 and Q2 for the sake of measuring the performance of TAPVerifier.
Q1: How is the accuracy of our privacy policy analysis module? More precisely, can it extract all useful sentences from privacy policies correctly? (Section V-C)
Q2: How is the accuracy of TAPVerifier when mapping privacy policy to different permissions? (Section V-D)

We also design experiments to answer the following five questions in order to assess whether TAPVerifier can enhance existing description based analysis system (i.e., AutoCog, Whyper, and CHABADA) by using privacy policy and code.
Q3: How many false alerts generated by the description analysis module (i.e. AutoCog) can be removed by using apps' and third-party libraries' privacy policies? (Section V-E1, V-E2)
Q4: How many false alerts generated by the description analysis module (i.e. AutoCog) can be removed by using code? (Section V-F)
Q5: How many false alerts generated by Whyper can be removed by using apps' privacy policies? (Section V-H)
Q6: Can privacy policies be used to explain the behaviors of malicious apps found by CHABADA [40]? (Section V-I)
The experimental results of Q3, Q4, Q5, and Q6 provide evidences to answer the two research questions proposed in Section I (i.e., RQ1 and RQ2).

A. Measurement and User Study of Privacy Policy

We first conduct a large-scale measurement to revisit the percentage of apps with privacy policy, and the result is shown in Table IV. After checking 4,202 apps, we find that 2,445 apps (i.e., 58.1%) provide privacy policy on Google play. Moreover, we observe that the percentage of apps with privacy policy increases with the number of installs. For instance, for the apps with 500,000 - 1,000,000 installs, only 44.6% of them provide privacy policy on Google play. In contrast, for the apps with 10,000,000 - 50,000,000 installs, 72.5% of them have privacy policy. Note that some apps may display the privacy policy inside the apps instead of posting it on Google play. For example, a recent study from Zimmeck et al. [90] checked 40 randomly selected apps that do not give privacy policy on Google play, and found that 17% (7/40) of them provide privacy policy links elsewhere (e.g., inside their apk files). TAPVerifier can also handle such apps after extracting their privacy policies from the apps.

Number of Installs	Percentage of Apps that Provide PP Link
500,000−1,000,000	44.6% (465/1043)
1,000,000−5,000,000	54.8% (965/1760)
5,000,000−10,000,000	65.9% (329/499)
10,000,000−50,000,000	72.5% (469/647)
50,000,000−100,000,000	83.6% (107/128)
100,000,000−500,000,000	86.9% (93/107)
1,000,000,000−5,000,000,000	94.4% (17/18)

TABLE IV: The percentage of apps with privacy policy.

We expect that all apps offered through Google play will include privacy policies soon, because Google recently updated its User Data Policy asking the apps in Google play to provide both a privacy policy and the secure handling of personal information. Moreover, Google has been removing apps from Google play if they do not comply with the new User Data Policy since Mar. 15, 2017 [64].

Then, we conduct a user study through Amazon Turk to examine how many users will read the privacy policy of the app by publishing a survey with 7 questions. 3 questions are about the background of the workers (i.e., age, gender, and education), and 4 other questions are related to smartphone and privacy policy, including: whether they use smartphones; the type of their phone (i.e., Android or iOS); whether they read privacy policy when downloading apps from app store; and whether they read the in-app privacy policy when using the app? We received 31 responses in 3 days. 64.5% (20/31) respondents are male, and the remaining respondents are female. Table V lists the age and education background of respondents. Most of them are 18-40 year old, and 67.7% (21/31) of them have at least associate degree. All of them use smartphones, where 87.1% respondents (i.e., 27/31) adopt Android phone while others select Apple phone.

Age	Number	Education	Number
<18	0	High School	1
18-40	24	Some College	9
40-60	6	Associate	3
>60	1	Bachelor	11
-	-	Graduate	7

TABLE V: The age and education background of the respondents.

Table VI lists the answers to the questions related to privacy policy: (1) ¡°Do you read the privacy policy when downloading app from app store? (denoted as "PP on app

store"); (2) "Do you read the in-app privacy policy when using the app?" (denoted as In-App PP). It shows that only 32.3% respondents will read the privacy policy when downloading an app, and 45.1% respondents will read the in-app privacy policy when using the app. A recent study [65] further shows that many factors will affect whether or not a user will read the privacy policy, including timing (when it is provided), channel (how it is delivered), modality (what interaction modes are used), and control (how are choices provided). Therefore, we could not rely on users' understanding of privacy policy to detect malware. Instead, we propose and develop TAPVerifier that can *automatically* process the privacy policy, bytecode, description, and permissions and conduct synthesized analysis on them to identify anomalies in apps.

Read PP or Not	PP on app store	In-App PP
Always	12.9%(4/31)	16.1%(5/31)
Sometimes	19.4%(6/31)	29.0%(9/31)
Never	67.7%(21/31)	54.9%%(17/31)
No smart phone	0.0%(0/31)	0.0%(0/31)

TABLE VI: The result of two privacy policy related questions: "Do you read the privacy policy when downloading app from app store?" "Do you read the in-app privacy policy when using the app?"

B. Data Set

The app data set in the previous version of this paper [83] contains 1,197 randomly selected apps. When creating the new data set, we remove 480 non-popular apps (downloaded for less than 100,000 times) because many of them (i.e., 132/480=27.5%) have been removed from Google play. Hence, we add 483 new popular apps, each of which has been downloaded for more than 100,000 times. The new data set contains 1,200 apps. To help researchers reproduce our work, we have upload the apk files, privacy policies, and descriptions of these 1,200 apps to the following URL: https://pan.baidu.com/s/1eRG0UIY.

C. Accuracy of TAPVerifier's Privacy Policy Analysis

To evaluate the accuracy of TAPVerifier's *privacy policy analysis module*, we randomly select 100 privacy policies from our sample set and split them into distinct sentences. Then, we divide these sentences into two groups: one contains useful sentences from which the collected information can be extracted and the other one contains useless sentences. The processing result of these sentences are manually verified by three researchers who are not authors of this paper. Before the manual verification, we explain to them the meaning of privacy policy and the definitions of useful sentence and useless sentence. Each sentence is checked by three researchers, and we use the majority opinion as the ground truth.

The *privacy policy analysis module* outputs 4,576 useful sentences and 5,501 useless sentences. The manual verification shows that among 4,576 useful sentences, 82 sentences are useless sentences (i.e., false positive), which account for 1.8%. Moreover, among 5,501 useless sentences, 104 sentences are useful sentences (i.e., false negative), which account for 1.9%. Thus, our module's precision is 98.2%, recall rate is 97.7%, and F-score is 97.9%.

$$Precision = \frac{TP}{TP + FP} = \frac{4494}{4494 + 82} = 98.2\% \quad (1)$$

$$Recall = \frac{TP}{TP + FN} = \frac{4494}{4494 + 104} = 97.7\% \quad (2)$$

$$F - score = \frac{2 * Precision * Recall}{Precision + Recall} = 97.9\% \quad (3)$$

Cause of false positives. One major cause is the hidden action executor in imperative sentences. For example, when processing the imperative sentence "*please read our summary of the changes*", although TAPVerifier successfully matches the verb "*read*", it decides that the action is executed by the app due to the lack of real action executor in this sentence, and therefore regards it as a useful sentence by mistake. However, the "*read*" action is conducted by the user.

Cause of false negatives. One major cause is due to the rare patterns that are not included in TAPVerifier. For example, the sentence "*you will be required to submit a valid user ID and password for authentication*" describes that the user will submit personal information to the server. However, in our semantic pattern defined for the user, we only consider the sentence whose $root\_word$ is in $VP_{provide}$. Since this sentence's $root\_word$ is "*require*", it is missed. To remove such false negative, we need to add the word "require" to VP_{allow} so that "be required to" will be matched and processed like "be allowed to".

D. Map Privacy Policy to Permissions

Table VII lists the number of various permissions that can be explained by privacy policy. For each permission, we first identify the number of apps that request it (i.e., column "Request Number"), and then count the number of apps whose privacy policies explain the necessity of the permission (i.e., column "PP Map Number (Percentage)"). We also calculate the accuracy of TAPVerifier (i.e., column "TAPVerifier precision").

Permission	Request Number	PP Map Number (Percentage)	TAPVerifier Precision
WRITE_SETTINGS	129	2 (1.6%)	100%
READ_CONTACTS	253	53 (**20.9**%)	89.8%
RECORD_AUDIO	148	7 (4.7%)	100%
WRITE_EXTERNAL_STORAGE	1004	75 (7.5%)	88.2%
WRITE_CONTACTS	76	3 (3.9%)	100%
ACCESS_COARSE_LOCATION	372	104 (**28.0**%)	92.0%
CAMERA	259	14 (5.4%)	93.3%
RECEIVE_BOOT_COMPLETED	343	-	-
GET_ACCOUNTS	657	15 (2.3%)	93.7%
READ_CALENDAR	39	3 (7.7%)	75.0%
ACCESS_FINE_LOCATION	365	106 (**29.0**%)	91.4%

TABLE VII: The number (percentage) of permissions whose necessity can be explained by privacy policy, and the precision of TAPVerifer for mapping privacy policy to permissions.

We can see that 3 permissions are explained by more than 20% privacy policies, including `READ_CONTACTS`, `ACCESS_COARSE_LOCATION`, and `ACCESS_FINE_LOCATION`. In Table VII, we use red font to emphasize them.

We also find 7 permissions explained by less than 20% privacy policies, including `WRITE_SETTINGS`,

RECORD_AUDIO, WRITE_EXTERNAL_STORAGE, WRITE_CONTACTS, CAMERA, GET_ACCOUNTS and READ_CALENDAR. Note that although many privacy policies contains word "account", we do not map them to GET_ACCOUNTS since they are about account registration or account information deletion. We currently do not consider the behaviors of web page and server.

False positives when mapping privacy policy to permission. After checking the errors, we find that the false positives are caused by the algorithm for computing semantic similarity (i.e., ESA). For example, since "credit card number" have a high semantic similarity with "sd card", it is mapped to permission WRITE_EXTERNAL_STORAGE. ESA transforms a text into a series of related words before calculating the semantic similarity value, and such wrong matching is unavoidable. We can use two methods to remove such false positive: one is selecting a higher threshold for ESA; the other is maintaining a black list of resources for each permission.

To measure the distribution of the contribution of each action to the definition of permissions, we further analyze the mappings between actions and permissions. A mapping between action AC and permission $PERM$ means that we can find one sentence $SENT$ that meets the following two conditions: 1) the syntactic structure of the sentence $SENT$ matches one of the semantic patterns of the action AC; 2) the sentence $SENT$ describes the collection, usage, storage, or disclosure of certain personal information protected by permission $PERM$. The first condition is checked according to the semantic patterns of each action, which are defined in Section III-B. The second condition is checked according to the mapping between personal information and permissions (Section IV-E1).

The result is shown in the Table VIII. We can see that the contribution of different actions to the definition of one permission are different. For example, for the permission READ_CONTACTS, the action DC (i.e., Data Collection) contains the largest number of mappings (i.e., 131). Another action UD (i.e., Data Utilization) only has 27 mappings. However, for the CAMERA permission, the action DC (i.e., Data Collection) and DU (i.e., Data Utilization) contain similar number of mappings (i.e., 17 and 16).

We also evaluate the usefulness of two new actions. The actions IS (i.e., Integrity & Security) and DAG (i.e., Data Aggregation) have 18 mappings in total. Although this number is smaller than that of other actions, some new sentences, which are missed by other actions, are discovered by the new actions. For example, the sentence "*We will take reasonable precaution to protect your information, contact information...*" is identified by the IS (i.e., Integrity & Security) action. Note that the pervious version of this study [83] does not include the IS action and therefore the sensitive information "contact information" included in this sentence is ignored.

E. Use privacy policy to remove false alerts generated by AutoCog

AutoCog raises an alert if a permission cannot be mapped to the description. However, some alerts are false alerts because the permissions can be mapped to the privacy policies or the permissions are over-claimed.

1) Using Apps' Privacy Policies: Table IX shows the number (percentage) of Autocog alerts we can remove by using app privacy policies through TAPVerifier(i.e., column "Removed Alert Num (Percentage)").

We can see that employing privacy policies can remove false alerts for all but the permission RECEIVE_BOOT_COMPLETED, which cannot be mapped to any privacy policy. But, the effectiveness of privacy policies is diverse for different permissions. For example, they can remove 44 false alerts for the permission WRITE_EXTERNAL_STORAGE. However, only 14 false alerts can be removed for the permission GET_ACCOUNTS. The reason is although many privacy policies contain account related sentences, the majority of them refer to account registration or sign up instead of accessing accounts in smartphone. Therefore we filter out such sentences.

2) Using Third-Party Libraries' Privacy Policies: We also use third-party libraries's privacy policies to remove false alerts. Since they cannot be mapped to all permissions, we show the result of relevant permissions in Table X. The result shows that such privacy policies can remove many false alerts due to the permissions WRITE_EXTERNAL_STORAGE, ACCESS_COARSE_LOCATION, GET_ACCOUNTS, and ACCESS_FALSE_LOCATION.

To evaluate the effectiveness of our two-stage approach for handling obfuscated third-party libraries mentioned in Section IV-C3), we randomly select 50 apps and utilize DeGuard's online service (i.e., http://apk-deguard.com) to process them before using TAPVerifier to handle them. Among the 50 apps, DeGuard recovers two new libraries in two apps after de-obfuscation, including the app "com.lucid_dreaming.awoken" that integrates Guava (Google Core Libraries for Java) and the app "com.nomanprojects.mycartracks" that embeds Google Gson (a Java library that can convert Java Objects into JSON representation). The class names of these two libraries are obfuscated in the raw apk files. DeGuard successfully uncovers their class names. By using the privacy policies of these two libraries, we can remove two permission alerts (GET_ACCOUNTS permission requested by these two apps) generated by AutoCog.

F. Use Code to Remove False Alerts

Table XI shows the number of false alerts that are generated by AutoCog but can be removed because they are over-claimed permissions. The column "AutoCog Alert Num" lists the number of AutoCog alerts for each permission. The column "Lib Use" shows the number of alert apps whose third library uses such permission. We maintain a white list of third party libraries. The column "Total Use" shows the number of alerted apps that use this permission in its code. The column "Over Claim" illustrates the number of alerted apps that over-claim certain permissions. The result clearly shows that many alerts can be removed after locating over-claim permissions.

Since the apps may use Java reflection to invoke APIs, we employ DroidRA [46] to process the 1,200 apps in our

Permission	DC	DS	DU	DA	DD	IS	UC	DP	DAG
WRITE_SETTINGS	3	2	0	1	0	0	0	1	0
READ_CONTACTS	131	23	27	24	12	5	1	23	0
RECORD_AUDIO	9	3	2	0	1	0	2	1	0
WRITE_EXTERNAL_STORAGE	29	56	2	0	0	2	4	0	0
WRITE_CONTACTS	36	21	1	0	0	2	0	1	0
ACCESS_COARSE_LOCATION	247	49	54	39	27	1	16	26	3
CAMERA	17	4	16	1	5	1	1	0	0
GET_ACCOUNTS	10	2	2	0	0	1	4	0	0
READ_CALENDAR	8	3	2	2	0	0	0	1	0
ACCESS_FINE_LOCATION	244	48	55	37	23	1	17	24	2

TABLE VIII: The number of mappings between different actions and different permissions. The first row lists the actions defined in Section III-B: DC (Data Collection), DS (Data Storage), DU (Data Utilization), DA (Data Access), DD (Data Disclose), IS (Integrity & Security), UC (User Consent), DP (Data Provision), and DAG (Data Aggregation).

Permission	AutoCog Alert	Removed Alert (Percentage)
WRITE_SETTINGS	107	2 (1.9%)
READ_CONTACTS	128	29 (22.6%)
RECORD_AUDIO	109	6 (5.5%)
WRITE_EXTERNAL_STORAGE	599	44 (7.3%)
WRITE_CONTACTS	59	2 (3.4%)
ACCESS_COARSE_LOCATION	266	76 (28.6%)
CAMERA	227	11 (4.8%)
RECEIVE_BOOT_COMPLETED	316	-
GET_ACCOUNTS	594	14 (2.4%)
READ_CALENDAR	30	3 (10.0%)
ACCESS_FINE_LOCATION	206	65 (31.6%)

TABLE IX: The number of alerts raised by AutoCog, and the number of alerts that can be removed through the analysis of privacy policy.

Permission	Autocog Alert	Removed Alert (Percentage)
WRITE_EXTERNAL_STORAGE	599	136 (22.7%)
ACCESS_COARSE_LOCATION	266	44 (16.5%)
GET_ACCOUNTS	594	10 (1.7%)
ACCESS_FINE_LOCATION	206	23 (11.2%)

TABLE X: The number (percentage) of alerts from AutoCog, which can be removed through the analysis of apps' and third-party libraries' privacy policies.

dataset. It reports that 879 apps use the reflection technique, and successfully recovers the invoked methods in 751 apps. It cannot recover the invoked methods in the other 128 apps (i.e., 879-751=128) because they only employ the reflection technique to construct object or obtain field (i.e., no methods are invoked via reflection).

Although 465 apps use the reflection technique to invoke framework APIs that can be located in the official document, only 52 of them call the APIs protected by permissions. For the APIs protected by permissions, the API *TelephonyManager.getDeviceId()* (protected by permission `READ_PHONE_STATE`) is the most frequently used one, which is called by 31 apps. For the APIs protected by the 10 permissions listed in Table VII, we find that 6 apps call camera related APIs and 2 apps call account related APIs via reflection. We do not find framework API invocations through reflection in the other 286 apps (i.e., 751-465=286) mainly due to two reasons. First, some APIs have been removed from the official document. For example, the app "aws.apps.networkInfoIi" calls the API *WifiManager.getWifiApState()*, which has been removed in the official document. Second, the invoked method is defined by the developer. For instance, the app "com.oristats.habitbull" integrates third-party library Flurry, which utilizes Java reflection to call the method *FlurryAdModule.getInstance()*.

G. Answers to RQs

We use the number of removed false alerts to measure the information that can be provided by privacy policy and bytecode for accessing the description-to-behavior fidelity. More precisely, we compare Table IX (and Table X) with Table XI to answer **RQ1** and **RQ2**.

Answer to **RQ1**: For some permissions, privacy policy can supply more information for accessing description-to-behavior fidelity, including:`READ_CONTACTS`, `WRITE_EXTERNAL_STORAGE`. For location related permissions, (i.e., `ACCESS_COARSE_LOCATION`, `ACCESS_FINE_LOCATION`), both privacy policy and bytecode can provide more information. Moreover, privacy policy cannot provide information related to the permission `REVEIVED_BOOT_COMPLETED`, but bytecode can achieve it.

Answer to **RQ2**: For some permissions, bytecode can provide more information for measuring description-to-behavior fidelity, including: `WRITE_SETTING`, `RECORD_AUDIO`, `WRITE_CONTACTS`, `CAMERA`, `RECEIVE_BOOT_COMPLETED`, `GET_ACCOUNTS`, `READ_CALENDAR`.

After getting the result of description analysis module, we can enhance it by combining the result of analyzing the app's privacy policy (Table IX), third-party libraries' privacy policies (Table X), and code (Table XI). The total number of false alerts that can be removed is shown in Table XII.

To analyze the cause of other alerts that cannot be removed by using privacy policy and bytecode level information, we randomly select 50 apps and upload them to VirusTotal to determine if they are malware or not. If the app is not reported as malware, we manually analyze the cause of these remained alerts. 19 apps are regarded as malicious apps by at least one anti-virus tool of VirusTotal. For example, TAPVerifier finds that the app "com.generamobile.headsoccer" requests three permissions (`ACCESS_FINE_LOCATION`,

Permission	Autcog Alert	Permission Used in Code		Over Claim (Percentage)
		Lib Use	Total Use	
WRITE_SETTINGS	107	4	50	57 (53.3%)
READ_CONTACTS	128	0	101	27 (21.1%)
RECORD_AUDIO	109	0	69	40 (36.7%)
WRITE_EXTERNAL_STORAGE	599	323	522	77 (12.9%)
WRITE_CONTACTS	59	0	49	10 (16.9%)
ACCESS_COARSE_LOCATION	266	106	202	64 (24.1%)
CAMERA	227	2	107	120 (52.9%)
RECEIVE_BOOT_COMPLETED	316	-	-	38 (12.0%)
GET_ACCOUNTS	594	17	308	286 (48.1%)
READ_CALENDAR	30	2	17	13 (43.3%)
ACCESS_FINE_LOCATION	206	66	160	46 (22.3%)

TABLE XI: Number of apps that are regarded as abnormal by AutoCog due to the over-claimed permissions.

Permission	AutoCog Alert	Removed Alert (Percentage)
WRITE_SETTINGS	107	59 (55.1%)
READ_CONTACTS	128	50 (39.1%)
RECORD_AUDIO	109	43 (39.4%)
WRITE_EXTERNAL_STORAGE	599	239 (39.9%)
WRITE_CONTACTS	59	11 (18.6%)
ACCESS_COARSE_LOCATION	266	158 (59.4%)
CAMERA	227	128 (56.4%)
RECEIVE_BOOT_COMPLETED	316	38 (12.0%)
GET_ACCOUNTS	594	303 (51.0%)
READ_CALENDAR	30	15 (50.0%)
ACCESS_FINE_LOCATION	206	115 (55.8%)

TABLE XII: Total number of AutoCog alerts and the number of alerts we can remove by using app privacy policy, third-party library privacy policy, and code at the same time.

Permission	Whyper Alert	Removed Alert (Percentage)
READ_CONTACTS	220	45 (20.5%)
RECORD_AUDIO	139	5 (3.6%)
READ_CALENDAR	29	2 (6.9%)

TABLE XIII: The number alerts raised by Whyper, and the number of alerts that can be removed through the analysis of privacy policy.

`ACCESS_COARSE_LOCATION`, and `WRITE_EXTERNAL_STORAGE`) without explaining them in description and privacy policy. This app is reported as malware by 7 of 55 anti-virus tools.

We manually analyzed the permissions of the 31 apps that do not receive any alerts from Virustotal, and have the following observations. 19 apps' descriptions and privacy policies only describe the apps' privacy related behaviors without covering the behaviors of the third-party libraries. Since the list of third-party libraries examined by TAPVerifier is by no means exhaustive, it missed some third-party libraries in those apps. Moreover, since some third-party libraries' privacy policies do not cover all behaviors of these libs, TAPVerifier identifies the inconsistency and raises an alert. The other 12 apps utilize sensitive permissions in their major code instead of third-party libraries, but they do not explain such behaviors in the descriptions or privacy policies. It may be due to the developers' carelessness. We believe that such problem may be mitigated by Google's new User Data Policy that requires each app in Google play to provide a privacy policy that should clearly list how user data is collected and handled [64].

H. Use Privacy Policy to Remove False Alerts Generated by Whyper

Whyper [54] is the pioneer detection system based on the description-to-behavior fidelity. It maps an app's description to three different permissions (i.e. `READ_CONTACTS`, `READ_CALENDAR`, and `RECORD_AUDIO`). By using the mapping between privacy policy and different permissions, TAPVerifier can also remove the false alerts generated by Whyper.

Table XIII shows the result. For `READ_CONTACTS`, TAPVerifier can remove 20.5% alerts. For `RECORD_AUDIO` and `READ_CALENDAR`, TAPVerifier can remove 3.6% and 6.9% alerts generated by Whyper, respectively.

I. Use Privacy Policy to Explain the Behaviors of Malicious Apps Found by CHABADA

We downloaded the dataset provided by CHABADA [6], which contains features and the clustering result of 26,332 apps. CHABADA labels 174 of these 26,332 apps as malicious ones. Among these 174 apps, we can successfully download the APK files of 12 apps that provide privacy policy. In the previous version of this paper, we can only download 11 apps because the other 2 apps are paid apps. But recently we find that the App 4 (i.e., "com.computertimeco.minishot.android") can be downloaded via the website: https://apps.evozi.com/apk-downloader/. Hence, we analyze it in this revised submission.

To check whether these apps are malicious or not, we upload their APK files to VirusTotal [12], which scans the APK files with 56-58 different anti-virus tools. If the app is reported as malicious by one or more anti-viruses, we regard the app as malware. 8 APK files are regarded as benign by all anti-virus tools. For the other 4 APK files (listed in Table XIV), each of them was regarded as a malicious app by only one anti-virus tool (i.e., other anti-virus tool do not raise alerts). Table XIV also lists the corresponding anti-virus tools that raise alerts and the detailed information.

Table XV shows the results of VirusTotal, CHABADA, and TAPVerifier, individually. VirusTotal raises alerts for App 1, 2, 3, and 12.

Using TAPVerifier to analyze these 12 apps, we find that the permissions requested by App 1 (`RECORD_AUDIO` and `WRITE_EXTERNAL_STORAGE`),

#	Apps	Anti-Virus tools that raise alert	Detail
1	com.appspot.swisscodemonkeys.steam	AegisLab	SUSPICIOUS
2	com.lonelycatgames.Xplore	Bkav	Android.Specapk.db.D8C0
3	es.cesar.quitesleep	K7GW	Spyware (004c0d821)
12	com.intsig.camscanner	Cyren	AndroidOS/GenBI.9FADF121!Olympus

TABLE XIV: The result from VirusTotal, including the anti-virus tools that raise an alert and the details from their reports.

#	Apk ID	Virus Total	CHABADA Alert	TAPVerifier Alert
1	com.appspot.swisscodemonkeys.steam	√	√	√
2	com.lonelycatgames.Xplore	√	√	
3	es.cesar.quitesleep	√	√	√
4	com.computertimeco.minishot.android		√	
5	com.droidhen.falldown		√	
6	com.netflix.mediaclient		√	
7	com.nubee.coinpirateS		√	
8	com.reverie.game.toiletpaper		√	
9	net.bible.android.activity		√	
10	org.mhgames.jewels		√	
11	si.modula.android.instantheartrate		√	√
12	com.intsig.camscanner	√	√	

TABLE XV: The results of VirusTotal, CHABADA, and TAPVerifier.

App 3 (`RECEIVE_BOOT_COMPLETED`), and App 11 (`CAMERA` and `WRITE_EXTERNAL_STORAGE`) are not explained by their descriptions and privacy policies.

For App 2 and App 12, TAPVerifier does not raise an alert whereas VirusTotal and CHABADA do. By checking the results of CHABADA and TAPVerifier, we find that CHABADA raises alerts for these two apps because they call network and device ID related APIs (e.g., *DefaultHttpClient.execute()* and *getDeviceId()*). However, since App 2 explains the usage of all requested permissions (including `INTERNET` and `READ_PHONE_STATE`) in its privacy policy, TAPVerifier regards it as a benign app, and therefore CHABADA raises a false alert for this app. In contrast, App 12 only explains the usage of Internet in its privacy policy without explaining the usage of device ID in either the privacy policy or description. Therefore, CHABADA raises an alert. TAPVerifier does not detect this apps because it does not take into account the permission `READ_PHONE_STATE`. If we include this permission in TAPVerifier, it will also raise an alert. Note that TAPVerifier examines the same number of permissions (i.e., 11) as AutoCog does whereas Whyper only studies three permissions. In future work, we will include more permissions in TAPVerifier.

For App 11, both CHABADA and TAPVerifier raise an alert but VirusTotal does not. By manually checking the app and the results of CHABADA and TAPVerifier, we find that both CHABADA and TAPVerifier raise a false alert because of the app's camera related behavior (*Camera.open()*). More precisely, this app explains this behavior in the description through the sentence "*Place the tip of your index finger on phone's camera*". CHABADA raises a false alert because it does not analyze the semantic meaning of individual sentences in the description. TAPVerifier leads to a false positive because its description analysis module (i.e., AutoCog) only extracts (verb, noun) pairs from the description without analyzing the prepositional phrase (i.e., "*on phone's camera*"). We will further enhance the description analysis module in future work.

Most apps detected by CHABADA may not be malicious ones (i.e., Apps 4-10), and neither VirusTotal nor TAPVerifier raises an alert for them. By examining the results of CHABADA, we find that it raises the alerts because these apps make network connections. It is worth noting that TAPVerifier does not consider the permissions related to establishing network connections, because TAPVerifier just focuses on the personal information. Similarly, neither AutoCog nor Whyper considers the permissions related to establishing network connections.

For verifying the results of CHABADA, we extend TAPVerifier's analysis to check whether these apps' descriptions and privacy policies can explain the behaviors of establishing network connections. The results show that three apps's behaviors can be explained by their privacy policies and one app's behavior can be explained by its description. In other words, these false positives can be removed by taking into account the apps' descriptions and privacy policies.

- App 5: com.droidhen.falldown. This app calls *WebView.<init>()* and *DefaultHttpClient.<init>()* to send data through internet. Its privacy policy contains the sentence "*Please note that certain features of the Services may be able to connect to your social networking sites to obtain additional information about you*", which can explain the use of internet.
- App 6: com.netflix.mediaclient. This app calls *DefaultHttpClient.<init>()* to access internet. Its description explains that "*Internet access and valid payment method are required to redeem offer*".
- App 7: com.nubee.coinpirates. This app calls network related APIs (e.g., *ConnectivityManager.startUsingNetworkFeature()* and *HttpURLConnection.connect()*). Its privacy policy explains that it may display advertisements through internet through the sentence "*The Company uses third-party service providers to display advertisements on many different websites available on the Internet*".
- App 8: com.reverie.game.toiletpaper. Although this app calls network related APIs (i.e., *ConnectivityManager.getActiveNetworkInfo()*), its privacy policy explains "*Please note that certain features of the Services may be able to connect to your social networking sites to obtain additional information about you*".

For the other three apps (i.e., App 4, 9, 10), the network behaviors found by CHABADA are not explained in the description and privacy policy.

VI. Threat to validity

Internal validity. Some threats may affect the effectiveness of TAPVerifier. First, when defining semantic patterns for

different actions, the verb sets are extracted from a corpus consisting of 500 privacy policies. We will increase the size of the corpus to improve the coverage of the verb sets.

Second, the static analysis module of TAPVerifier can only identify a limited number of third-party libraries. We will add more third-party libraries into the system so that we could remove more false alerts by using the libraries' privacy policies. Moreover, TAPVerifier does not conduct dynamic analysis, native code analysis, and dynamic code loading analysis, which can be exploited by malware to evade the detection. To avoid this threat, we will integrate dynamic analysis systems (e.g., DroidScope [81]), native code analysis (e.g., NDroid [57]), and dynamic class loading analysis (e.g., [55]) for improving TAPVerifier's performance in future work.

Third, TAPVerifier relies on PScout [20] to determine the APIs/URIs relevant to privacy-sensitive permissions. Although its website only provides the mapping between API/URI and permissions for up to Android 5.1.1, TAPVerifier can also work on the latest version of Android by using either of the two following methods. First, we can utilize PScout to process the latest version of Android framework (https://github.com/zd2100/PScout) for getting the new mapping. Second, besides PScout, Erik Derr et al. proposed another system [21] that conducts static analysis on Android framework to identify the mapping between API/URI and permissions. The authors have published their result (including the result of Android 6.0) on the website http://www.axplorer.org/. We could employ this result in TAPVerifier for handling the latest version of Android.

In Table XVI, we quantify the changes on the APIs associated with privacy-sensitive permissions in different versions of Android [20]. It shows that the number of changed APIs is relatively small. For example, when the system is upgraded from Android 3.2.2 to Android 4.0.1, only 24 new APs are identified, which account for 14.8% (24/162). When the system is upgraded from Android 4.0.1 to Android 4.1.1, only 3 new APIs are found, which account for 1.8% (3/164).

Permission	3.2.2	4.0.1		4.1.1	
	Total	Total	New	Total	New
WRITE_SETTINGS	18	19	3	21	2
READ_CONTACTS	21	28	7	27	0
RECORD_AUDIO	7	7	0	7	0
WRITE_EXTERNAL_STORAGE	4	4	0	4	0
WRITE_CONTACTS	21	27	6	27	0
ACCESS_COARSE_LOCATION	21	21	0	21	0
CAMERA	3	3	0	3	0
RECEIVE_BOOT_COMPLETED	2	0	0	0	0
GET_ACCOUNTS	16	24	8	25	1
READ_CALENDAR	0	7	0	7	0
ACCESS_FINE_LOCATION	22	22	0	22	0

TABLE XVI: Number of APIs associated with privacy-sensitive permissions in different versions of Android according to the results from PScout [20]. The column ¡§Total¡¨ refers to the total number of APIs protected by each permission. The column ¡§New¡¨ refers to the number of APIs that are not found in the previous version of Android.

Fourth, since the description analysis module only extracts (verb, noun) pairs from the description, TAPVerifier cannot identify the nouns included in other places (e.g., prepositional phrase). We will enhance this module by taking into consideration the whole sentence in future work.

Finally, since the current version of TAPVerifier does not analyze all permissions (i.e., only 11 permissions are considered in this paper), it may miss some permissions that are requested by the app without explanation in either description or privacy policy. We will add more permissions to TAPVerifier to overcome this threat. Moreover, in future work, we will enhance TAPVerifier to process in-app privacy policy by first using static analysis to identify the in-app privacy policy. More precisely, we could first recover the GUI structure of each activity in an app, and then check the content of the TextView widget and the text associated with each button. If the text contains the phrase "privacy policy", we extract the URL link provided by the developer.

We use the app "com.alienmanfc6.wheresmyandroid" as an example to illustrate this procedure. After obtaining the GUI structure of the starting activity, we find that the layout of this activity (defined in `layout/setup_welcome.xml`) includes a *TextView* widget (`id=@id/setup_terms_textview`, `text=@string/terms_agreement`). The string `terms_agreement` is defined in the `values/strings.xml` file: "*By using this app you agree to the terms of service and privacy policy. You can read the full agreement online at <a href = "http://wheresmydroid.com/terms.html" > wheresmydroid.com/ terms.html*". Since this string contains phrase "*privacy policy*", we extract the URL "http://wheresmydroid.com/terms.html" as the in-app privacy policy related URL.

External validity. The major threat to external validity is the correctness of the ground-truth when we check the useful sentences and the permissions identified by TAPVerifier. Currently, we ask three researchers to check the processing results. In future work, we will invite more people with experiences in handling privacy policy to create corpus for verification.

Another threat is the quality of the privacy policy, which will affect the number of alerts that can be removed by TAPVerifier. Since such investigation deserves another paper, we will explore the quality of these privacy policies and measure its impact on TAPVerifier in future work.

The last threat is the correctness of Virustotal (Section V-I). Since Virustotal employs a large number of antivirus to scan the app, some antivirus may generate incorrect result. To decrease this threat, we suggest the developers follow the latest tips [4] [2] to provide clear description and privacy policy for additional check.

VII. Related work

A. Text Analysis for Mobile Security

The description of apps have been analysed for mobile security, such as CHABADA [40], Whyper [54], and AutoCog [58]. CHABADA is different from Whyper and AutoCog since it finds abnormal APIs by comparing the app with other apps in the same cluster. However, Whyper and AutoCog find suspicious permissions by checking the descriptions of apps. ACODE [75] first finds APIs/URIs used in code, and then uses keywords search technique to find related sentences in the description of the app.

The reviews of apps can also be used. AUTOREB [44] searches keywords in review and then uses a trained spare linear support vector to map the review to security-related behaviors. Slavin et al. detected the sensitive APIs called in code but are not mentioned in privacy policy [66]. Different from TAPVerifier, they manually extract data collection phrases from many privacy policies and do not consider retrieving sensitive information through content providers.

This paper is an extended version of our previous conference paper [83]. Compared with the previous work, we extended the contents from the following aspects. First, we enhance the semantic patterns for privacy policy (Section III-A) by adding two new kinds of actions into the data-flow model, including data aggregation, integrity & security. By integrating two new actions into TAPVerifier, we can conduct a fine-grained analysis on privacy policy. In Section III-B, we propose a mechanism to automatically build up verb sets by analyzing existing privacy policy corpus. We also define semantic patterns for the new actions, which can identify the corresponding sentences from privacy policies.

Second, we improve TAPVerifier from several aspects (Section IV). We enhance the negation analysis so that it can process sentences with complex structure (Section IV-B6). We improve the static analysis algorithm in Section IV-C so that TAPVerifier can identify the source APIs/URIs used by third-party libraries. This information from third-party libraries' privacy policies is used to remove the false alerts of the description analysis module. TAPVerifier also integrates DroidRA to find out the APIs called via reflection technique (Section IV-C3). In Section IV-E1, we include a new algorithm to map the privacy policy to different permissions. By considering both the verbs of personal information and the verbs in permission, the new algorithm can obtain a more accurate mapping. We also change the algorithm to remove false alerts by using third-party libraries' privacy policies in Section IV-E2. Since the algorithm used in our previous paper [83] does not consider the behaviors of third-party libraries, some errors may appear. For the new algorithm, only if a third-party library uses some permissions in code and declares it in the lib privacy policy, the corresponding alert can be removed.

Third, we add many new experiments in Section V. We measure how many apps with privacy policies and conduct a user case study to study how many users will read the privacy policy (Section V-A). We include a new Section V-D to show the number of permissions that can be explained by using the privacy policy. In this section, we also analyze the mappings between actions and permissions. When checking the number of alerts we can remove by using third-party libraries' privacy policies (Section V-E2), we utilize the de-obfuscation tool DeGuard to recover the class names of obfuscated third-party libraries. After removing the alerts of AutoCog with privacy policy and description, we analyze the cause of the remaining alerts (Section V-G). We also include a new Section V-H to show that TAPVerifier can use privacy policy to remove the false alerts generated by the system Whyper. Finally, we add a new Section V-I to demonstrate that TAPVerifier can use privacy policy to enhance the performance of another existing description based system CHABADA.

B. Mobile Malware Detection

Stowaway [35] utilizes automated testing techniques to build up the map between Android permissions and APIs. Then it disassembles the dex file to detect over-claimed permissions. To increase code coverage, PScout [20] performs static reachability analysis on framework to build up mapping between API calls and permissions. Bartel et al. [24] combine Class Hierarchy Analysis (CHA) and field-sensitive static analysis (Spark) to analyze Android permissions. VetDroid [87] conducts dynamic analysis to understand how the resources protected by permissions are accessed and utilized. Felt et al. find that the unrestricted intent-based ICC mechanisms can be used to conduct intent spoofing attack [36]. They develop a system ComDroid to examine permission-requiring intents. Wei et al. [76] perform long term study on the permission usage of the entire Android ecosystem. They find that an increasing number of apps are violating the principle of least privilege. Wu et al. [78] find out that 85.78% of preloaded apps in are over-claimed and 66.40% are due to vendor customizations. Backes et al. [21] report that PScout generates some false mappings since it does not conduct in-depth analysis on the application framework. Currently, we select PScout to build the mapping between permission and APIs/URIs. In the future, we will replace PScout with [21], [24] to improve the correctness of our system. More discussion about Android permission analysis can be found in [80], [72], [70], and [63].

Various features that can be extracted by static analysis have been proposed to detect mobile malware. DroidSIFT [85] represents app with weighted contextual API dependency graphs and then uses Naive Bayes classifier to identify malware family. AppContext [82] extracts security-sensitive API calls and their context information as features, and then uses SVM to determine whether an action is legitimate or not. Apposcopy [37] uses inter-component call graph to represent the control flow property and then uses static taint analysis to get the data flow property to represent an app. Some other static analysis systems focus on detecting the privacy leaks in app. AAPL [49] uses the conditional data flow analysis and joint data flow analysis to find data leakages in apps. It also leverages the similar apps recommended by Google Play to remove false alarms. SUPOR [42] uses NLP techniques to identify sensitive input fields and conducts taint analysis on the data originated from sensitive input fields to detect privacy leakage. UI-Picker [52] extracts all text labels in UI and sends them to a supervised learning classifier to determine the input is sensitive or not.

DroidScope [81] intercepts certain events and parses kernel data structure to reconstruct OS-level and Java-level semantic views, which enables it to monitor communication between different components such as Java components, native components, Android Java Framework and the Linux kernel. Enck et al. developed a dynamic taint system TaintDroid [34]. TaintDroid is an extension to the Android OS that provides real time monitoring of user data and detects data leaving the mobile device. As traditional dynamic taint checking systems such as TaintDroid and DroidScope uncover information flows

through JNI, Chenxiong et al. developed a system NDroid to get all low-level instructions of Android by instrumenting QEMU based on which to conduct taint propagation [57]. CopperDroid [71] instrumented the Android emulator to collect the system calls invoked by the apps running inside the emulator. The binder-related system calls with their real-time arguments are sent to the unmarshalling Oracle which runs alongside CopperDroid to automatically deserialize Binder communications. Uranine [62] instruments the Android apps to detect privacy leakage in real time. The major advantage of Uranine is that it does not require system modification and source code of apps. In order to detect malware effectively, DroidNative [13] performs static analysis on both the dynamically loaded native code and the bytecode to build up Control Flow Graph (CFG) and annotates the CFG with Malware Analysis Intermediate Language (MAIL) patterns.

C. Privacy Policy Analysis

Privee performs coarse-grained analysis on privacy policies by classifying them into six categories [89]. Costante et al. performed a sentence-level analysis to determine what information will be collected by a web site [32]. It divides the action verbs in three groups and defines five semantic patterns in an ad-hoc manner. Massey et al. used topic model to extract key words from 2,061 policy documents [14] and proposed a taxonomy that can be applied to many domains [50]. Massey et al. [51] compared two taxonomies created by Anton [15], [16] and Solove [67] [68]. Recently, HMM is used to align the sections in privacy policies according to their contents [47], [59]. Breaux et al. evaluate the time and resource required for crowdsourcing the tasks of analyzing privacy policies [28]. Moreover, they proposed Eddy to find conflicts between privacy policies [27]. Sunyaev et al. checked top 600 most commonly used mobile health apps and found that 30.5% them had privacy policies [69]. At the same time, average privacy length was 1755 words [69]. Balebako et al. [22] conduct experiment and survey to indicate that showing the privacy notice during app use significantly increased recall rates over showing it in the app store. Since not all apps in Google Play provide privacy policies, a system named AutoPPG is developed [84]. AutoPPG can leverage static analysis to find sensitive API/URIs used in code and then utilize NLP technique to generate privacy policy sentences for developers.

VIII. Conclusion

We propose using privacy policy and bytecode to enhance malware detection systems that rely on checking the description-to-behavior fidelity in apps. More precisely, we propose a novel data flow model for analyzing privacy policy, and develop TAPVerifier for carrying out investigation of privacy policy, bytecode, description, and permissions, and conducting the cross-verification among them. The experimental result through real apps shows that our privacy policy analysis module can achieve 97.7% recall and 98.2% precision. Moreover, TAPVerifier can remove up to 59.4% false alerts of the state-of-the-art systems.

IX. Acknowledgment

We thank the anonymous reviewers for their quality reviews and suggestions. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E, 152279/16E), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), Hong Kong RGC Project (No. CityU C1008-16G), the HKPolyU Research Grants (G-YBJX), and the National Natural Science Foundation of China (No. 61602371),

References

[1] The state of mobile app privacy policies. http://goo.gl/2A18Wj, 2012.
[2] Guide to developing an app privacy policy. https://goo.gl/woQSVz, 2014.
[3] Google play unofficial python api. https://github.com/egirault/googleplay-api, 2015.
[4] Write an app store description that excites with these 5 tips. https://goo.gl/OW3ljO, 2015.
[5] Android library statistics: Social sdks. http://goo.gl/Bsth3A, 2016.
[6] Chabada: Checking app behaviors against app descriptions. https://www.st.cs.uni-saarland.de/appmining/chabada/, 2016.
[7] Natural language toolkit. http://www.nltk.org/, 2016.
[8] Upload and distribute apps. https://support.google.com/googleplay/android-developer/answer/113469?hl=en, 2016.
[9] Android development tools. https://goo.gl/hRTlMW, 2017.
[10] Beautiful soup. http://goo.gl/0Lh7Dk, 2017.
[11] A brief overview of shimple. https://github.com/Sable/soot/wiki/A-brief-overview-of-Shimple, 2017.
[12] Virustotal for android. http://goo.gl/uBLt3E, 2017.
[13] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *computers & security*, 65, 2017.
[14] A. Anton, J. Earp, Q. He, W. Stufflebeam, D. Bolchini, and C. Jensen. Financial privacy policies and the need for standardization. *IEEE Security & Privacy*, 2(2), 2004.
[15] A. I. Anton and J. B. Earp. A requirements taxonomy for reducing web site privacy vulnerabilities. *Requirements Engineering*, 9(3):169–185, 2004.
[16] A. I. Anton, J. B. Earp, and A. Reese. Analyzing website privacy requirements using a privacy goal taxonomy. In *Proc. Requirements Engineering*, 2002.
[17] AppBrain. Top 80 popular ad libraries. http://goo.gl/GBhXOi, 2015.
[18] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. NDSS*, 2014.
[19] 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.
[20] K. Au, Y. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proc. ACM CCS*, 2012.
[21] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *Proc. USENIX Security*, 2016.
[22] R. Balebako, F. Schaub, I. Adjerid, A. Acquisti, and L. Cranor. The impact of timing on the salience of smartphone app privacy notices. In *Proc. SPSM*, 2015.
[23] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. SOAP*, 2012.
[24] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *Trans. TSE*, 40(6):617–632, 2014.
[25] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev. Statistical deobfuscation of android applications. In *Proc. CCS*, 2016.
[26] D. Blei. Probabilistic topic models. *Communications of the ACM*, 55(4), 2012.
[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*, 19(3), 2014.

[28] T. Breaux and F. Schaub. Scaling requirements extraction to the crowd: Experiments on privacy policies. In *Proc. IEEE RE*, 2014.

[29] 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.

[30] D. Cer, M. Marneffe, D. Jurafsky, and C. Manning. Parsing to stanford dependencies: Trade-offs between speed and accuracy. In *Proc. LREC*, pages 1628–1632, 2010.

[31] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applications with permission event graphs. In *Proc. NDSS*, 2013.

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

[33] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, 1995.

[34] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. OSDI*, 2010.

[35] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. ACM CCS*, 2011.

[36] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proc. Usenix Security*, 2011.

[37] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proc. ACM FSE*, 2014.

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

[39] R. Girju, A. Badulescu, and D. Moldovan. Learning semantic constraints for the automatic discovery of part-whole relations. In *Proc. NAACL*, 2003.

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

[41] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proc. ACM MobiSys*, 2012.

[42] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *Proc. USENIX Security*, 2015.

[43] IDC Corporate. Android and ios squeeze the competition, swelling to 96% of the smartphone operating system market for both 4q14 and cy14. http://goo.gl/Lo9cwq, Feb. 2015.

[44] D. Kong, L. Cen, and H. Jin. Autoreb: Automatically understanding the review-to-behavior fidelity in android applications. In *Proc. CCS*, 2015.

[45] 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.

[46] 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.

[47] F. Liu, R. Ramanath, N. Sadeh, and N. Smith. A step towards usable privacy policy: Automatic alignment of privacy statements. In *Proc. COLING*, 2014.

[48] Lookout Inc. 2014 mobile threat report. http://goo.gl/8mD8tz, 2015.

[49] 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.

[50] A. Massey, J. Eisenstein, A. Anton, and P. Swire. Automated text mining for requirements analysis of policy documents. In *Proc. IEEE RE*, 2013.

[51] A. K. Massey and A. I. Antón. A requirements-based comparison of privacy taxonomies. In *Proc. RELAW*, 2008.

[52] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *Proc. USENIX Security*, 2015.

[53] B. OConnor and M. Heilman. Arkref: A rule-based coreference resolution system. *arXiv:1310.1975*, 2013.

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

[55] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proc. NDSS*, 2014.

[56] C. Qian, X. Luo, Y. Le, and G. Gu. Vulhunter: toward discovering vulnerabilities in android applications. *IEEE Micro Mag.*, 35(1), 2015.

[57] C. Qian, X. Luo, Y. Shao, and A. T. Chan. On tracking information flows through jni in android applications. In *Proc. DSN*, 2014.

[58] 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.

[59] R. Ramanath, F. Liu, N. Sadeh, and N. Smith. Unsupervised alignment of privacy policies using hidden markov models. In *Proc. ACL*, 2014.

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

[61] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic large-scale dynamic analysis of android applications. In *Proc. CODASPY*, 2013.

[62] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen. Uranine: Real-time privacy leakage monitoring without system modification for android. In *International Conference on Security and Privacy in Communication Systems*, 2015.

[63] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, et al. * droid: Assessment and evaluation of android application analysis tools. *ACM Computing Surveys (CSUR)*, 2016.

[64] J. Richard J. Caira and T. Ey. Heads up, app developers: Google is getting serious about privacy and data security in apps. https://goo.gl/eyzDAL, 2017.

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

[66] 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. http://goo.gl/E13Fst, 2015.

[67] D. J. Solove. A taxonomy of privacy. *University of Pennsylvania law review*, 2006.

[68] D. J. Solove. *Understanding privacy*. Harvard University Press, 2008.

[69] A. Sunyaev, T. Dehling, P. L. Taylor, and K. D. Mandl. Availability and quality of mobile health app privacy policies. *Journal of the American Medical Informatics Association*, 2015.

[70] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4), 2017.

[71] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proc. NDSS*, 2015.

[72] D. J. Tan, T.-W. Chua, V. L. Thing, et al. Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 2015.

[73] Trademob. How to write an app description and drive more download. http://goo.gl/q1mJ2k, 2013.

[74] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proc. CASCON*, 1999.

[75] T. Watanabe, M. Akiyama, T. Sakai, H. Washizaki, and T. Mori. Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps. In *Proc. SOUPS*, 2015.

[76] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *Proc. ACSAC*, 2012.

[77] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. http://goo.gl/f7Ci0k, 2014.

[78] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proc. CCS*, 2013.

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

[80] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, et al. Toward engineering a secure android ecosystem: a survey of existing techniques. *ACM Computing Surveys (CSUR)*, 2016.

[81] L. Yan and H. Yin. Droidscope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis. In *Proc. USENIX Security*, 2012.

[82] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behavior under contexts. In *Proc. ICSE*, 2015.

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

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

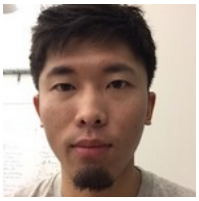
[85] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proc. ACM CCS*, 2014.

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

[87] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. ACM CCS*, 2013.
[88] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. NDSS*, 2012.
[89] S. Zimmeck and S. M. Bellovin. Privee: An architecture for automatically analyzing web privacy policies. In *Proc. USENIX Security*, 2014.
[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.

Le Yu received the bachelor's and master's degrees in information security from the Nanjing University of Posts and Telecommunications. He is currently pursuing the Ph.D. degree with the Department of Computing, The Hong Kong Polytechnic University. His current research focuses on mobile security.

Xiapu Luo received the Ph.D. degree in computer science from The Hong Kong Polytechnic University. He was a Post-Doctoral Research Fellow with the Georgia Institute of Technology. He is currently a Research Assistant Professor with the Department of Computing and an Associate Researcher with the Shenzhen Research Institute, The Hong Kong Polytechnic University. His current research focuses on smartphone security and privacy, network security and privacy, and Internet measurement.



Chenxiong Qian received the bachelor's degree in software engineering from Nanjing University. Currently he is a third-year Ph.D. student in the School of Computer Science at the Georgia Institute of Technology, advised by Prof. Wenke Lee and Prof. Bill Harris. This work was done when he was a research assistant in the Department of Computing, Hong Kong Polytechnic University. Chenxiong studies system security and privacy. He is particularly interested in using program analysis to solve system security and privacy problems.

Shuai Wang received the bachelor's degree in computer science from Southwest University, China. He was a a research assistant with the Department of Computing, The Hong Kong Polytechnic University.

Hareton K. N. Leung received the Ph.D. degree in computer science from the University of Alberta. Currently he is an associate professor and director of the Laboratory for Software Development and Management in the Department of Computing, the Hong Kong Polytechnic University. His research interests include software testing, project management, risk management, quality and process improvement, software metrics, and e-health.