Revisiting the Description-to-Behavior Fidelity in Android applications

Le Yu, Xiapu Luo §, Chenxiong Qian, Shuai Wang
Department of Computing, The Hong Kong Polytechnic University
The Hong Kong Polytechnic University Shenzhen Research Institute
{cslyu, csxluo, csxqian, csswang}@comp.polyu.edu.hk

***Abstract*—Since more than 96% of mobile malware targets on Android platform, various techniques based on static code analysis or dynamic behavior analysis have been proposed to detect malicious applications. As malware is becoming more complicated and stealthy, recent research proposed a promising detection approach that looks for the inconsistency between an application's permissions and its description. In this paper, we revisit this approach and find that using description and permission will lead to many false positives. Therefore, we propose employing app's privacy policy and its bytecode to enhance description and permission for malware detection. 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. We propose a novel data flow model for analyzing privacy policy, and develop a novel 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 code level information removes 8.1%-65.5% false positives of existing systems based on description and permission.**

I. Introduction

The massive success of app economy poses lucrative and profitable targets for attackers. It has been showed that the number of mobile malware has jumped 75% in 2014 [1]. Moreover, while Android has taken up 81.5% market share with millions of applications (or simply apps) [2], 96% mobile malware targets Android [1].

Many detection systems based on static analysis [3]–[8] and/or dynamic analysis [9]–[14] 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 have 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 [15]–[17]. 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 [15], AutoCog [16]) profile an app's expected behaviors by extracting the semantic meaning from its description, and characterize an app's behaviors by examining the permissions required by the app. CHABADA identifies abnormal sensitive API usages from apps expected to provide similar functionality according to their descriptions [17].

§The corresponding author.

Although their results are encouraging, there still lacks of a systematic study on assessing an app's description-to-behavior fidelity because of two reasons. First, since descriptions on Google Play have character limit, they cannot detail all behaviors, thus leading to false positives (i.e., "hide" security-related behaviors). Since 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 [18]), we argue that privacy policy should be taken into account for profiling what an app advertises. Second, since developers may over-claim permissions [19], using permissions to represent an app's behaviors will result in false positives (i.e., "overclaim" security-related behaviors). Since the app's behaviors are determined by its bytecode, we argue that the app's bytecode should be considered for characterizing what an app behaves.

Therefore, in this paper, we leverage both an app's privacy policy and its bytecode to revisit 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 challenging 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 non-trivial to automatically extract their meanings [20]. We propose a novel privacy policy data flow model to create semantic patterns, which are used to automatically identify actions in privacy policy. Then, we extract an app's expected behavior by employing information extraction (IE) and natural language processing (NLP) techniques. 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 develop TAPVerifier, a ***T***ext-based ***AP***plication ***V***erification system. TAPVerifier integrates the analysis of these four kinds software artifacts together and performs the cross-verification automatically. Our major contributions include:

1) We propose a novel privacy policy data flow model for defining semantic patterns to infer what sensitive information an app will collect from its privacy policy.

Based on these patterns, we build up a novel privacy policy analysis module to automatically extract collected personal information from privacy policy files.

2) We propose TAPVerifier, a new system for analyzing privacy policy, bytecode, description, and permission, and conducting cross-verification among them. Moreover, we have implemented TAPVerifier in 6,381 lines python code and 11,078 lines java code.
3) Experiment result shows that compared with description, privacy policies are more likely to describe privacy-related behaviours regardless of the categories. Furthermore, using privacy policy and bytecode removes 8.1%-65.5% false positives generated by approaches based on the description and permission analysis.

The reminder of this paper is organized as follows. We formulate the problem in detail and describe the motivating examples in SectionII. SectionIII details how we identify semantic patterns for analyzing privacy policies. SectionIV describes the design and the implementation of TAPVerifier. SectionV presents the extensive evaluation results and observations. After discussing TAPVerifier's limitations and introducing our future work in SectionVI, we introduce related work in SectionVII and conclude the paper in SectionVIII.

II. Preliminaries

A. Background

1) Privacy policy: When publishing an app in Google play market, developers will provide other information to help user learn more about the app, such as, description, privacy policy, screenshots, to name a few [21]. Description is like an ad for promoting the app and attracting more users to download this app [22]. To make the description appealing to users, the app's most relevant features are put in it. Privacy policy is a hybrid statement, and contains different aspect about information collection, such as what information would be collected, how information is used, how can user access their personal information, and etc. [23].

2) Android: Each app has an APK file that contains `dex` file, manifest file(`AndroidManifest.xml`), resource file and other supporting files. An app's executable is a `dex` file which can be disassembled for further analysis. Android use permissions to limit the access to sensitive data or feature on the device. If an app wants to use some feature protected by permissions, it must declare corresponding permission in app's manifest file.

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 [24]. 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 [24].

B. Motivating Example

We use the app "com.tinymission.dailyyogafree" to illustrate how to use privacy policy to remove false positives resulted from the insufficiency of description, and another app "com.ilspl.mahavir" to demonstrate how to use code analysis to remove false positives due to the overclaimed permissions.

> **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. 1. Snippet of com.tinymission.dailyyogafree's privacy policy.

Use privacy policy to explain the necessary of permission. Fig.1 lists part of the app's privacy policy. "Daily Workout (means "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`. Therefore, the privacy policy can explain the necessity of requesting this permission.

Use privacy policy to remove false positives. When analyzing this app's description, `Autocog` [16] cannot locate any sentence that can explain why the permission `ACCESS_FINE_LOCATION` is needed, and therefore it will raise a permission alert. However, since the privacy policy can explain the necessary, such alert is a false positive.

Use code to remove false positives. The app, "com.ilspl.mahavir", requests permission `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION` without explaining it in description. `AutoCog` generates an alert. However, our code analysis finds that this app does not use any location related APIs. Therefore, such alert is a false positive.

III. Semantic Patterns for Privacy Policies

A. Data Flow Model for Privacy Policies

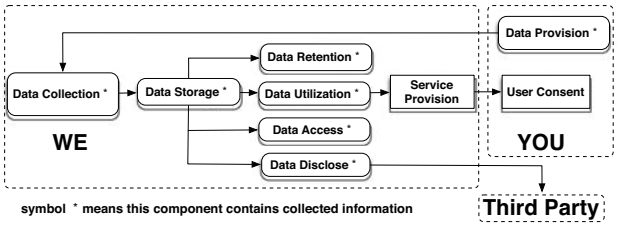

Fig. 2. Data Flow Model for Privacy Policies.

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. The extraction of collected information is based on identifying useful sentences. Other sentences are regarded as "useless", and will not be analysed. For example, although 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 is 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 for identifying useful sentences and extracting specific information. More precisely, we create a data flow model for major components in privacy policies. This model describes how users' information is processed and transmitted. Although privacy policy writer may use different kinds of sentences for different parts of this model, we extract the patterns based on this model so that all sentences related to private information can be identified.

As shown in Fig. 2, 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 blocks. The actor **We** may refer to the app itself, the developer/owner of this app, or the service provider. **We** can collect private information of user from the app. The actor **You** refers to the user of an app or service, and **You** can provide information to developer through registering account or other channels. The actor **Third Party** collaborates with **We**, such as Ad providers, and may receive the information collected by **We**. Our model is general and extensible, and it does not require a privacy policy to have all blocks. Moreover, if a new action is identified, we can easily add it to the model.

The model in Fig.2 illustrates how information flows from one actor to another and how it is handled by different actions. We detail each action as follows because they will guide us to define semantic patterns in Section III-B.

▷ **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. As the information will first be obtained before **We** can use it, the sentences about this action will disclose the collected information, such as, "*We may use your location information to display advertisements for businesses*".

▷ **Data Retention**. Privacy policies usually mention how long the collected information will be kept by **We** and how **You** can access or manipulate the information. Therefore, the corresponding sentences will describe the information, for instance, "*We'll retain information you store on our Services for as long as we need it to provide you the Services. If you delete your account, we'll also delete this information.*"

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

▷ **Data disclose**. It explains what, when, 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.*".

▷ **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**. Since sometimes **We** will ask **You** to provide certain information directly, the relevant sentences will present the details, for instance, "*you will be asked to provide us with your phone number, name and a photo (name and photo are not mandatory) and to allow us access to your mobile device's address book.*".

The **Service Provision** action usually indicates the service or data provided to the user briefly. Since an app's description provides more similar information than **Service Provision**, we do no analyze this action.

B. Semantic Patterns

We define semantic patterns according to the data flow model shown in Fig. 2. More precisely, we first find out the 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 developer can use any one in sentence.

We employ three approaches to select verbs. First, we select 23 verbs from 35 most commonly-used verbs in privacy policies summarized by [25]. We ignore the remaining 12 verbs, such as "employ", "advise", "aggregate", etc., because they do not fit any actions in Fig.2. Second, we manually examine five privacy policy templates and extract 44 verbs from them [26] [27] [28] [29] [30]. Since many developers employ such templates to create privacy policies, these verbs are representative. Third, we used the tool WordNet [31] to find out the synonyms of the verbs found in above steps. Wordnet is a lexical database of English where synonyms of nouns, verbs, and adjectives, adverbs are collected according to their meaning similarity.

We finally collect 109 verbs and classify them to 11 sets according to their semantic meaning. Tab. I lists three sample verbs for each category. 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**)	storage, reserve, log,...
6	VP_{use} (**Data Utilization**)	use, process, link,...
7	$VP_{disclose}$ (**Data Disclose**)	share, sell, disclose,...
8	$VP_{rentain}$ (**Data Retention**)	retain, maintain, delete,...
9	VP_{allow} (**All actions**)	allow, disallow, permit,...
10	$VP_{provide}$ (**All actions**)	provide, supply, offer,...
11	$VP_{consent}$ (**User Consent**)	consent, agree, assent,...

TABLE I
COMMON VERBS AND THEIR RELATED ACTIONS.

According to the common sentence structures, we define nine general semantic patterns as listed in Tab. II. We use VP_* to represent the verb set (1,3,5,7,8,10) in Tab. I and use VP_*^{pass} to indicate the corresponding *passive* voice of the verbs. To ease the presentation of semantic patterns, we put *resource* in the place where the collected information will appear. Note that the semantic patterns for each action in Fig. 2 are derived from these nine general semantic patterns. By replacing VP_* in Tab. II with different verb sets defined in Tab. I, we get the semantic pattern for each action. We detail them in the following paragraphs.

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 would collect your personal information*" (Pattern_DC 1) or "*your personal information would be collected*" (Pattern_DC 2). They may ask the permission to collect some information, for example, "*you allow us to collect your personal information*" (Pattern_DC 4) or "*we are allowed to collect your personal information*" (Pattern_DC 5). Pattern_DC 3 indicates a special class of descriptive sentences that enumerate individual collected information, for instance, "*the information we collect include: your name, address, age*". The subject and the object form the part-whole relation [32], where name, address, and age are part of the collected information.

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

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}.

Data Access. Its common semantic patterns include:
Pattern_DA 1: $sbj\ VP_{access}\ resource$
Pattern_DA 2: $resource\ VP_{access}^{pass}$
Pattern_DA 3: $sbj\ VP_{allow}\ obj\ to\ VP_{access}\ resource$
Pattern_DA 4: $sbj\ VP_{allow}^{pass}\ to\ VP_{access}\ resource$
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 personal information is accessible*". 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 would limit access to personal to third party*".

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

Pattern_DU 1,2 are similar to Pattern_DA 1,2, but the verb are in VP_{use}. Pattern_DU 3, 4 allow "us" to use user's personal information.

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

For data disclose action, we also defined four semantic patterns, just like patterns defined in **Data Utilization**, but VP_{use} is replaced with $VP_{disclose}$.

Data Retention. Its common semantic patterns include:
Pattern_DR 1: $sbj\ VP_{retain}\ resource$
Pattern_DR 2: $resource\ VP_{retain}^{pass}$

Pattern_DR 1 denotes that the retention of data has a time limit. Pattern_DR 2 indicates that if the time reaches the time limit or the user sends a request, all these data need to be removed.

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

This pattern matches sentences like "*you are consent to the collection of your personal information*".

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 your name and address to register a account*".

IV. APPVERIFIER

A. Architecture

Fig.3 shows the architecture of TAPVerifier, which takes in the privacy policy, description, and APK file of an app as input. The *privacy policy analysis module* (Section IV-B) processes

#	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	$sbj\ VP_{consent}\ to\ something$	You are consent to the collection of your personal information.

TABLE II
GENERAL SEMANTIC PATTERNS.

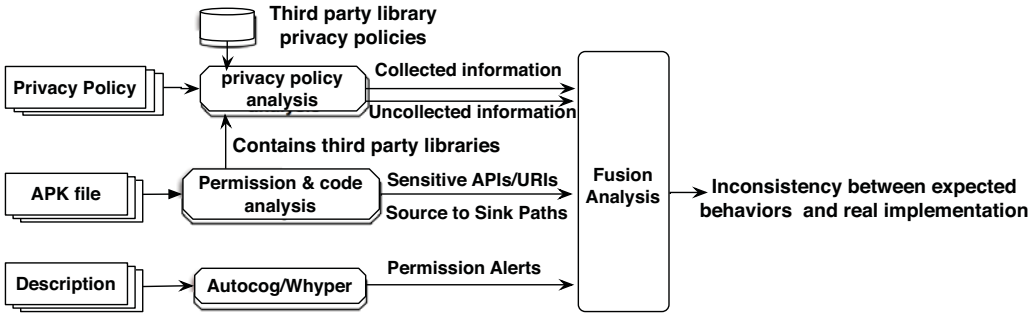

Fig. 3. TAPVerifier's Architecture

the privacy policy file and outputs a list of information that 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 [33]. APGs are stored in a graph database. Then the module will look for sensitive APIs and third party libraries, and conduct depth first traversal to identify paths from sources to sinks.

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.

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 requested permissions. It also inspects the code to remove the false alerts due to over-claim permissions.

B. Privacy Policy Analysis

1) Overview: We employ IE and NLP techniques to process privacy policies. As shown in Fig. 4, the procedure has the following major steps. The pre-processing step (SectionIV-B2) extracts text from the privacy policy file in HTML format and splits it into distinct sentences. The syntactic parsing step (SectionIV-B3) parses distinct sentences and generates syntactic trees and typed dependencies. The syntactic trees and typed dependencies are stored in database.

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

2) Pre-processing: Since each privacy policy is saved in HTML format, we use Beautiful Soup [34] to extract the text content from the HTML file. 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) 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 [35].

3) Syntactic Parsing: For each sentence, TAPVerifier employs Stanford Parser [36] to parse 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.5 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 that 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

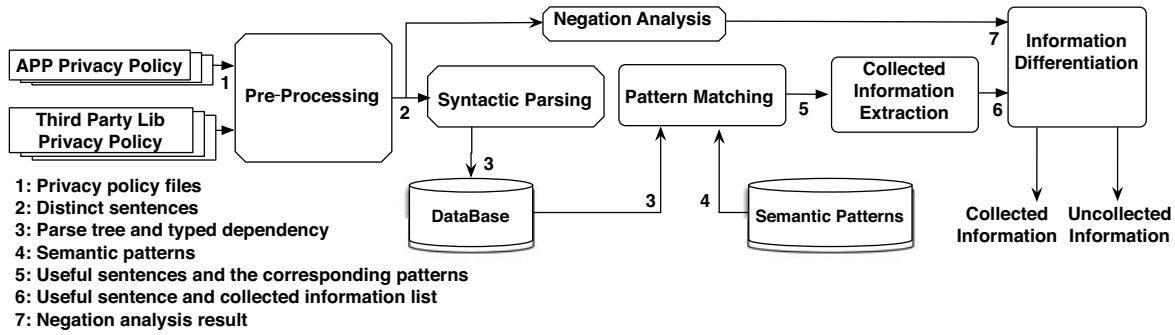

Fig. 4. The procedure of privacy policy analysis

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.5, NP means noun phrase, VP denotes verb phrase, PRP indicates pronoun, VB represents verb, and NN expresses noun. The typed dependencies offer 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.

Parse Tree

```
(ROOT
 (S
  (NP (PRP we))
  (VP (MD would)
   (VP (VB use)
    (NP
     (NP (PRP$ your) (NN location))
     (, ,)
     (NP
      (NP (NN account) (NN information))
      (SBAR
       (WHADVP (WRB when))
       (S
        (NP (PRP you))
        (VP (VBP use)
         (NP (PRP$ our) (NN app)))))))))
 (. .)))
```

Typed dependencies

```
nsubj(use-3, we-1)
aux(use-3, would-2)
root(ROOT-0, use-3)
poss(location-5, your-4)
dobj(use-3, location-5)
nn(information-8, account-7)
appos(location-5, information-8)
advmod(use-11, when-9)
nsubj(use-11, you-10)
rcmod(information-8, use-11)
poss(app-13, our-12)
dobj(use-11, app-13)
```

Fig. 5. Parse tree and typed dependencies

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 syntactic information extracted from syntactic parsing step. Those sentences that cannot be mapped to any semantic patterns will be removed. The useful sentences found in this step and their corresponding semantic patterns are the input of the collected information extraction step. The pattern matching algorithm is shown in Algorithm 1, where if a sentence matched any one of the 9 general semantic patterns defined in Tab. II, it is a useful sentence.

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

Due to page limit, 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 Tab. II) and passive voice (like general semantic pattern 2 in Tab. II). To match the general semantic pattern 1 and 2, we look up the dependency relationship, find the word that has a "root" dependency relation with the node $Root - 0$. Note that, in the following section, we call this word $root\_word$. Since sample sentences 1 and 2 in Tab. II 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**. So after getting $root\_word$, in line 2, we look up Tab. I to find its corresponding verb set and determine the sentence's action.

We use different methods to extract the collected information from passive voice sentences and active voice sentences. For an active voice sentence (e.g., Tab.II sample sentence 1), the collected information is the object of $root\_word$, while in a passive voice sentence (e.g., Tab. II 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 dependency relation list ("auxpass" means "passive auxiliary").

Note that, 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 $root\_word$ belongs to $VP_{provide}$, this sentence's action executor should not be **We** because only the 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

Input: str_sent : sentence to match; $Dep\_Relations$: Typed Dependency Relation list.
Output: 1,2,3,..,8,9: General semantic pattern number; 0: Match fail.

$root\_word = getWord(root, Root - 0)$
$cate = getVerbCate(root\_word)$
if $cate == VP_*$ **then**
  // try to match pattern 1,2
  if $len(getWord(auxpass, root\_word)) == 0$ **then**
    return 1;
  end
  return 2;
else if $cate == VP_{contain}$ **then**
  // try to match pattern 3
  for *sbj in* $getWord(nsubj, root\_word)$ **do**
    for *mod_word in* $getWords(rcmod, sbj)$ **do**
      if $getVerbCate(mod\_word) == VP_{collect}$ **then**
        return 3;
      end
    end
  end
else if $cate == VP_{allow}$ **then**
  // try to match pattern 4,5
  $passive\_words = getWord(auxpass, root\_word)$
  for $verb \in getWord(xcomp, root\_word)$ **do**
    if $getVerbCate(verb) == VP_*$ **then**
      if $len(passive\_words) == 0$ **then**
        return 4;
      end
      return 5;
    end
  end
else if $cate == ADJ_{access}$ **then**
  // try to match pattern 6
  return 6;
else if *"able to" in* str_sent || *"keep ability to" in* str_sent **then**
  // try to match pattern 7
  if *"able to" in* str_sent **then**
    $verb = getWordAfter(str\_sent, "able\ to")$
  else
    $verb = getWordAfter(str\_sent, "keep\ ability\ to")$
  end
  if $getVerbCate(verb) == VP_{access}$ **then**
    return 7;
  end
else if $cate == VP_{access-control}$ && *"access to" in* str_sent **then**
  // try to match pattern 8
  return 8;
else if $cate == VP_{consent}$ **then**
  // try to match pattern 9
  $verb = getWord(prep\_to, root\_word)$
  if $VP_* inverb$ **then**
    return 9;
  end
else
  return 0; // all pattern match fail, return 0;
end

Algorithm 1: Semantic Pattern Match

words, once a general semantic pattern is determined, TAPVerifier looks for the corresponding *resource* as shown in Tab.II in the parse tree. Note that we do not extract the noun phrases in the conditional clauses. Moreover, we remove stop words for improving the accuracy.

To improve the performance, we adopt ARKref [37] to conduct the co-reference resolution. If one pronoun denotes collected information, the corresponding noun will be added to the list of collected information.

6) Negation Analysis: When performing the negation analysis, we consider negative determiners (e.g., "*no*", "*neither*"), negative adjectives (e.g., "*unable*", "*improper*"), negative nouns (e.g., "*nobody*", "*none*"), and verbs (e.g., "*prevent*", "*prohibit*", "*forbid*") with negative connotation, and adverbs (e.g., "*hardly*", "*scarcely*", "*barely*") [38]. If the subject or main verb related words contain negative word, we regard this sentence as a negative one.

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 analyze their privacy policies individually. To prepare the database for popular third party libraries' privacy policies, we download the SDK and the privacy policies of top 83 Ad libraries listed in [39], 9 social libraries [40], and 24 most commonly used development tools [41]. 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 [33], 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 [42] to recover the `dex` file. By parsing the `AndroidManifest.xml` file, TAPVerifier finds out all components and the required permissions. Then, we use Soot [43] to transform the Dalvik code in `dex` file to the intermediate representation *Shimple*. Based on the class hierarchy and the intermediate representation, we create an Android property graph (APG) [33] that integrates abstract syntax tree (AST), interprocedure control-flow graph (ICFG), method call graph (MCG), and system dependency graph (SDG) of the app.

Since Android is event-driven, there exists implicit control flow transitions through the Android framework [44]. To improve the precision of our static analysis system, we leverage the transitions found by EdgeMiner [44] to enhance our MCG, ICFG, and SDG. 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. FlowDorid [46] is the state-of-art static taint analysis system. The source to sink paths found by FlowDroid are also included when building SDG.

To find third-party libraries used in the app, we maintain a white list which contains class name prefixes of common third party libraries. When enumerating all class names contained in an app, if a class name has the same the prefix as a library, we think that the corresponding third-party library is used.

2) Traversals: After building graphs for each app, we perform two kinds of traversals to collect information.

(1) 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.

Currently, TAPVerifier has 128 source functions, which can get the following information: device ID, IP address, cookie, location, account, contact, account, calendar, telephone number, software version, video, and audio, running task, application information. We also select sensitive 9 URI strings and 476 URI fields for getting information from content provider. More will be included in future work.

(2) Path from source to sink.

The existence of a source-to-sink path means that the app collects some information and delivers to sinks. Apart from the source functions and related URIs used in **(1)**, we select 54 sink functions which transfer data to SMS, log, file, and Internet. Since not all source-to-sink paths are executable at runtime, we use Joogie [47] to detect all infeasible methods of apps. If a path contains infeasible method, we remove it from the result.

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 [48]. 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.

D. Description Analysis

Since AutoCog handles more permissions with better performance than Whyper [16], 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.

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, this sentence is mapped to the corresponding permission. After processing all sentences and all permissions, if one permission cannot be mapped to any sentence in the description, AutoCog raises an alert [16].

E. Fusion Analysis

Since existing systems like AutoCog and Whyper have contrasted an app's description and its permissions, the fusion analysis module focuses on privacy policy and code. We describe the methods in next section, and present the experimental results and insights in Section V.

1) Use privacy policy to explain the necessary of permission: To map privacy policy to permission, 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 [48] 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 [49], 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 0.5.

2) Use privacy policy to remove false positives: When developers request some permissions and mention such behaviors in an app's privacy policy instead of its description, we can leverage privacy policy to remove false alerts resulted from 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 privacy policy and then removes them, thus improving the accuracy of description analysis module.

3) Use code to remove false positives: Since apps may claim more permissions than they need [19], 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 6,381 lines python code and 11,078 lines java code. We have also developed a crawler in 1,334 line python codes to automatically fetch apps' APK files, descriptions, privacy policies from Goolge play market [50].

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

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-B)

Q2: How many false alerts generated by description analysis module can be removed by using privacy policy? (Section V-C1, V-C2)

Q3: How many false alerts generated by description analysis module can be removed by using code? (Section V-D)

A. Data Set

We randomly select 1,197 apps from Google Play as our dataset. All these selected apps have descriptions and privacy policies in English. We use TAPVerifier to process the descriptions, and privacy policies of these apps. The APK's code level information are all put into graph database.

B. 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 1197 samples 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 people 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 sentence (i.e., false positive), which account for 1.7%. Moreover, among 5,501 useless sentences, 104 sentences are useful sentences (i.e., false negative), which account for 1.9%. Then, our module's precision is 98.2%, recall rate is 97.7%, F-score is 97.9%.

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 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".

C. Use privacy policy to remove false alerts

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: Tab. III shows the number (percentage) of Autocog alerts we can remove by using app privacy policies through TAPVerifier(i.e., column "PP Map Num (Percentage)") and the precision of TAPVerifier when mapping privacy policy to permission (i.e., column "TAPVerifier Precision").

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 109 false alerts for the permission `WRITE_EXTERNAL_STORAGE`. However, only 29 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.

Permission	AutoCog Alert Num	PP Map Num (Percentage)	TAPVerifier Precision
WRITE_SETTINGS	74	6 (8.1%)	85.7% (6/7)
READ_CONTACTS	98	26 (26.5%)	83.8 % (26/31)
RECORD_AUDIO	89	5 (5.6%)	100.0% (5/5)
WRITE_EXTERNAL_STORAGE	582	109 (18.7%)	88.6 % (109/123)
WRITE_CONTACTS	53	17 (32.1%)	73.9 % (17/23)
ACCESS_COARSE_LOCATION	243	51 (21.0%)	98.0 % (51/52)
CAMERA	201	30 (15.0%)	93.8 % (30/32)
RECEIVE_BOOT_COMPLETED	184	-	-
GET_ACCOUNTS	463	29 (6.3%)	100.0% (29/29)
READ_CALENDAR	31	6 (19.4%)	85.7 % (6/7)
ACCESS_FINE_LOCATION	203	40 (19.7%)	93.0 % (40/43)

TABLE III
NUMBER (PERCENTAGE) OF AUTOCOG ALERTS WE CAN REMOVE THROUGH APP PRIVACY POLICY AND THE PRECISION OF APPVERIFER WHEN MAPPING PRIVACY POLICY TO PERMISSION.

False positives when mapping privacy policy to permission. As you can see in Tab. III, the precision of TAPVerifier is not 100%. After checking the errors, we find that such false positives are caused by 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.

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 Tab.IV. The result shows that such privacy policies can remove many false alerts due to the permissions `ACCESS_COARSE_LOCATION`, `ACCESS_FALSE_LOCATION`, `READ_CONTACTS` and `WRITE_EXTERNAL_STORAGE`.

Permission	Autocog Alert	PP Map Num (Percentage)
READ_CONTACTS	98	54 (55.1%)
RECORD_AUDIO	89	1 (1.1%)
WRITE_EXTERNAL_STORAGE	582	39 (6.7%)
ACCESS_COARSE_LOCATION	243	152 (62.5%)
ACCESS_FINE_LOCATION	203	133 (65.5%)

TABLE IV
NUMBER (PERCENTAGE) OF AUTOCOG ALERTS WE CAN REMOVE THROUGH THE PRIVACY POLICIES OF THIRD PARTY LIBRARIES.

D. Use code to remove false alerts

Tab. V 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" 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 Num" 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.

E. Answers to RQs

We can use the number of removed false alerts to measure the information that can be provided by privacy policy and bytecode for accessing description-to-behavior fidelity. More precisely, we compare Tab.III and Tab.V 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_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 `REVEIVED_BOOT_COMPLETED` related information, but bytecode can.

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

VI. Threats to Validity

Construct validity. Some threats will affect the construct validity of our system, for example, TAPVerifier only considers verbs of five privacy policy templates. To mitigate this threat, we will analyze more templates in future work.

Our static taint analysis system cannot verify paths dynamically, and hence dead code may cause false alerts. To reduce this threat, we integrate the tool Joogie [47] to remove paths that contain infeasible methods. At the same time, the static analysis module cannot identify the APIs used by app and the APIs used by injected malware. Since ResDroid [51], a repackaged app detection tool, can find the major packages by using PageRank algorithm, we will integrate it into TAPVerifier in future work.

When we use the privacy policies of third party libs to remove false positives of `AutoCog`, our system do not check the user of the resources. This may affect the correctness of the fusion analysis module. For example, if the resource is used by one third party lib, but the corresponding resource is only declared by another third party lib. In this case, the alert should not be removed.

The last threat in system design is that we do not analyze the action of each permission. For example, if the app requires permission `WRITE_CONTACTS`, but the privacy policy declares "*we will read contacts*", then our system will not send alert. However, the privacy policy and permission are inconsistent. To avoid such errors, we will enhance the permission and code analysis module by extracting the actions of permissions.

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

External validity. The major threat to external validity is the representativeness of our dataset. Currently, we randomly select 1,197 sample apps, and wewill add more apps into the data set in future work.

VII. Related work

A. Text Analysis for Mobile Security

The descriptions of apps are analysed for mobile security. Whyper [15] and AutoCog [16] extract word pairs from description and then map them to permissions requested by apps. The difference is that Whyper's semantic model is built by manually analyzing API documents while AutoCog creates it by conducting statistical analysis on a large number descriptions.

CHABADA [17] combines description and the called APIs to find abnormal apps. It uses LDA to extract topic words from descriptions for grouping apps into different clusters. Then it identifies abnormal apps with abnormal API usages in the same cluster. Note that, CHABADA is different from Whyper and AutoCog since CHABADA 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 [52] first finds APIs/URIs used in code, and then it uses keywords search technique to find related sentences in the description of the app. The reviews of apps can also be used. AUTOREB [53] searchs 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 [54]. Different from TAPVerifier, they manually extract data collection phrases from many privacy policies and do not consider retrieving sensitive information through content providers.

B. Mobile Malware Detection Based on Static Analysis

Various features that can be extracted by static analysis have been proposed to detect mobile malware. DroidSIFT [4] represents app with weighted contextual API dependency graphs and then uses Naive Bayes classifier to identify malware family. AppContext [55] 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 [6] 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

Permission	#Autcog Alert	Permission Used in Code		Over Claim Num (Percentage)
		Lib Use	Total Use	
WRITE_SETTINGS	74	4	35	39 (52.7%)
READ_CONTACTS	98	0	82	16(16.3%)
RECORD_AUDIO	89	0	65	24 (27.0%)
WRITE_EXTERNAL_STORAGE	582	240	518	64 (11.0%)
WRITE_CONTACTS	53	0	47	6(11.3%)
ACCESS_COARSE_LOCATION	243	103	188	55 (22.6%)
CAMERA	201	0	119	82(40.8%)
RECEIVE_BOOT_COMPLETED	184	-	-	29 (15.8%)
GET_ACCOUNTS	463	7	229	234(50.5%)
READ_CALENDAR	31	2	17	14 (45.2%)
ACCESS_FINE_LOCATION	203	77	164	39 (19.2%)

TABLE V
NUMBER OF APPS AUTOCOG ALERT BUT ARE OVER-CLAIMED PERMISSIONS IN FACT. THE COLUMN "OVER CLAIM NUM" REFERS TO NUMBER OF APPS AUTOCOG ALERT BUT DO NOT USE THE PERMISSION IN CODE.

static analysis systems focus on detecting the privacy leaks in app. AAPL [56] 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 [57] 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 [58] extracts all text labels in UI and sends them to a supervised learning classifier to determine the input is sensitive or not.

C. Privacy Policy Analysis

Privee performs coarse-grained analysis on privacy policies by classifying them into six categories [20]. Costante et al. performed a sentence-level analysis to determine what information will be collected by a web site [59]. 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 [23] and proposed a taxonomy that can be applied to many domains [60]. Recently, HMM is used to align the sections in privacy policies according to their contents [61], [62]. Breaux et al. evaluate the time and resource required for crowdsourcing the tasks of analyzing privacy policies [63]. Moreover, they proposed Eddy to find conflicts between privacy policies [64]. Since not all apps in Google Play provide privacy policies, a system named AutoPPG is developed [65]. 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 involving 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 8.1%-65.5% false alerts generated by original 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), the National Natural Science Foundation of China (No. 61202396), the HKPolyU Research Grant (No. G-UA3X), the Hong Kong ITF (No. UIM/285), and China Postdoctoral Science Foundation (No. 2015M582663).

REFERENCES

[1] Lookout Inc., "2014 mobile threat report," http://goo.gl/8mD8tz, 2015.
[2] 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.
[3] K. Chen, N. Johnson, V. Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and D. Song, "Contextual policy enforcement in android applications with permission event graphs," 2013.
[4] 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.
[5] 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.
[6] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proc. ACM FSE*, 2014.
[7] 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.
[8] 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.
[9] 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.
[10] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Security*, 2012.
[11] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *Proc. DSN*, 2014.
[12] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic large-scale dynamic analysis of android applications," in *Proc. CODASPY*, 2013.
[13] 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.

[14] K. Tam, S. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," 2015.

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

[16] 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.

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

[18] "The state of mobile app privacy policies," http://goo.gl/2A18Wj.

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

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

[21] "Upload and distribute apps," https://support.google.com/googleplay/android-developer/answer/113469?hl=en.

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

[23] A. Anton, J. Earp, Q. He, W. Stufflebeam, D. Bolchini, and C. Jensen, "Financial privacy policies and the need for standardization," *IEEE Security & Privacy*, vol. 2, no. 2, 2004.

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

[25] A. Anton and J. Earp, "A requirements taxonomy for reducing web site privacy vulnerabilities," *Requirements Engineering*, vol. 9, no. 3, 2004.

[26] FreePrivacyPolicy.com, "Create a free custom privacy policy," http://www.freeprivacypolicy.com/, 2015.

[27] termsfeed.com, "Sample privacy policy template," https://termsfeed.com/blog/sample-privacy-policy-template/, 2015.

[28] seqlegal.com, "Privacy policy," http://www.seqlegal.com/free-legal-documents/privacy-policy, 2015.

[29] upcounsel.com, "Privacy policy template," https://www.upcounsel.com/privacy-policy-template, 2015.

[30] shopify.com, "Personalized privacy policy generator," https://ecommerce.shopify.com/policy-generator, 2015.

[31] "Wordnet," http://wordnet.princeton.edu/.

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

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

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

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

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

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

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

[39] AppBrain, "Top 80 popular ad libraries," http://goo.gl/GBhXOi, 2015.

[40] "Android social sdks," http://goo.gl/Bsth3A.

[41] "Android development tools," https://goo.gl/hRTlMW.

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

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

[44] 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.

[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] 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.

[47] S. Arlt, P. Rümmer, and M. Schäf, "Joogie: From java through jimple to boogie," in *Proc. SIGPLAN*, 2013.

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

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

[50] "Google play unofficial python api," https://github.com/egirault/googleplay-api, 2015.

[51] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proc. ACSAC*, 2014.

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

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

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

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

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

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

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

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