AutoPPG: Towards Automatic Generation of Privacy Policy for Android Applications

Le Yu, Tao Zhang, Xiapu Luo*, Lei Xue
Department of Computing, The Hong Kong Polytechnic University
The Hong Kong Polytechnic University Shenzhen Research Institute
{cslyu,cstzhang,csxluo,cslxue}@comp.polyu.edu.hk

ABSTRACT

A privacy policy is a statement informing users how their information will be collected, used, and disclosed. Failing to provide a correct privacy policy may result in a fine. However, writing privacy policy is tedious and error-prone, because the author may not well understand the source code, which could be written by others (e.g., outsourcing), or does not know the internals of third-party libraries without source codes. In this paper, we propose and develop a novel system named *AutoPPG* to automatically construct correct and readable descriptions to facilitate the generation of privacy policy for Android applications (i.e., apps). Given an app, AutoPPG first conducts various static code analyses to characterize its behaviors related to users' private information and then applies natural language processing techniques to generating correct and accessible sentences for describing these behaviors. The experimental results using real apps and crowdsourcing indicate that: (1) AutoPPG creates correct and easy-to-understand descriptions for privacy policies; and (2) the privacy policies constructed by AutoPPG usually reveal more operations related to users' private information than existing privacy policies.

Categories and Subject Descriptors

K.4.1 [**Computing Milieux**]: Public Policy Issues—*Privacy*

Keywords

mobile applications; privacy policy; information extraction; program analysis; document generation

1. INTRODUCTION

As smartphones have become an indispensable part of our daily lives, users are increasingly concerned about the privacy issues on the personal information collected by various apps. Although the Android system can list all permissions required by an app before its installation, such approach may not help users understand the app's behaviors, especially those related to users' private information, because of the lack of precise and accessible descriptions [27, 34]. Alternatively, developers can provide privacy policies to their apps for informing users how their information will be collected, used, and disclosed [4]. Actually, the Federal Trade Commission (FTC) suggested mobile developers to prepare privacy policies for their apps and make them easily accessible through the app stores [6]. Other major countries or areas (e.g., European, Australia, etc.) have privacy laws for requiring developers to add privacy policies [7].

Failing to provide a correct privacy privacy may result in a fine. For instance, the social networking app "Path" was fined $800,000 in 2013 because it collected and stored users' personal information without declaring the behaviors in its privacy policy [8]. As another example, FTC filled a complaint[1] against a popular app named "Brightest Flashlight Free" in 2014 because its privacy policy does not correctly reflect how it uses personal data [25].

However, writing privacy policy is tedious and error-prone because of many reasons. For example, the author of a privacy policy may not well understand the app's source code, which could be outsourced, or the precise operation of each API invoked. Moreover, the developer may not know the internals of the integrated third-party libraries, which usually do not provide source code. Existing approaches for generating privacy policies cannot solve these issues because few of them analyze code and therefore they heavily rely on human intervention, such as answering questions like "what personal information do you collect?". Moreover, Privacy Informer [26] could only generate privacy policies for apps created by App Inventor[2] instead of normal apps.

To facilitate the generation of privacy policy, in this paper, we propose a novel system named *AutoPPG* to automatically construct correct and readable descriptions for an app's behaviors related to personal information. It is nontrivial to develop AutoPPG because of several challenging issues. First, how to automatically map APIs to private information? Note that other similar studies (e.g., AppProfiler [34]) did it manually. By exploiting the Google Java style followed by the Android framework [9] and information extraction techniques [24], we propose a new algorithm (i.e., Algorithm 1 in Section 3.1) to address this issue.

*The corresponding author

SPSM'15, October 12, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2808117.2808125.

[1] http://goo.gl/7pjSgC
[2] http://appinventor.mit.edu

Second, how to profile an app's behaviors related to private information through static analysis? By leveraging the app property graph(APG) [30], we design various graph traversals to answer questions like (Section 3.2): does the app collect private information? If yes, does the host app or any third-party library collect such information? Does the app notify users when collecting the information? Will the app store the information locally or send it to a remote server? Third, how to construct correct and readable descriptions? By analyzing 500 apps' privacy policies and employing natural language processing (NLP) approaches, AutoPPG can generate correct and accessible descriptions (Section 3.3 and 3.4), which are validated through crowdsourcing-based experiments (Section 4.3).

In summary, this paper makes the following contributions:

1. We propose AutoPPG, a novel system that automatically constructs correct and readable descriptions to facilitate the generation of privacy policy for Android applications. To our best knowledge, AutoPPG is the first system that can construct such information by leveraging static code analysis and NLP techniques.

2. We tackle several challenging issues for developing AutoPPG, including automatically mapping APIs to private information, profiling an app's behaviors related to private information through static analysis, and constructing correct and readable descriptions. These techniques can be applied to solve other research problems, such as malware detection.

3. We have implemented AutoPPG and perform careful experiments with real apps to evaluate its performance. The experimental results show that AutoPPG can construct correct and easy-to-understand descriptions for privacy policy and the privacy policies resulted from AutoPPG usually reveal more operations related to users' private information than existing privacy policies.

The rest of this paper is structured as follows. Section 2 presents the background. Section 3 details the design and implementation of AutoPPG while Section 4 describes the experimental results. Section 5 introduces the threats to validity of this study. After introducing the related work in Section 6, we conclude the paper and introduce the future work in Section 7.

2. BACKGROUND

2.1 Privacy policy

According to [7], a privacy policy may contain five kinds of information: (1) contact and identity information; (2) the classification of personal information that the app wants to collect and use; (3) the reasons why the data is needed; (4) authorization information for the public to third parties and persons; (5) the right that users have. Since (1) and (5) concern users' information and rights and (3) explain why certain data is needed, they cannot be extracted by analyzing an app's code. Therefore, AutoPPG focuses on generating statements for (2) and (4).

As an example, Fig. 1 shows part of the app *com.macropinch.swan*'s privacy policy. It contains the identity information shown in the top part, and the contact information shown in the bottom part, both of which are in dashed line rectangles. Note that such information belongs to (1) and AutoPPG cannot generate. The sentences in the red box present which information will be collected and the statements in the blue box describe how the information will be disclosed. Note that such information belongs to (2) and (4), respectively, and therefore AutoPPG can create them.

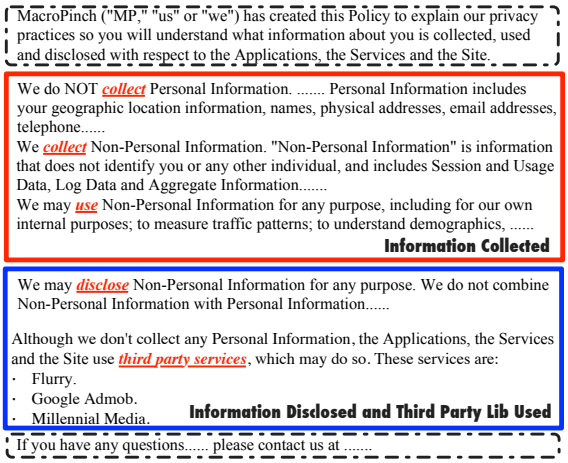

Figure 1: Privacy Policy Example: Information com.macropinch.swan can collect, use, and disclose

2.2 The sentence structure

Figure 2: The structure of a general sentence in privacy policy.

A general sentence in privacy policy contains three key parts, including executor, action, and resource. Other parts such as condition (*i.e.*, when this action happens), purpose (*i.e.*, why "we" do this thing), and target (*i.e.*, whom "we" send this information to) are optional.

- Executor is the entity who collects, uses and retains information. If the subject is "we", like the sentence shown in Fig. 2, the behaviour is executed by the app; if the subject is the third party library, then this information will be disclosed to third party library.
- Action refers to what the executor does, such as "collect" in Fig. 2.
- Resource is the object on which an action operates. In Fig. 2, the resource is "your location information".

In this paper, *private information* refers to the private data that can be obtained by an app from smartphones, such as "location", "contact", "device ID", "audio", "video", etc. We treat them as resource and they serve as the object of an action verb, because AutoPPG currently just generates sentences in active voice.

3. AUTOPPG

As shown in Fig. 3, AutoPPG consists of three modules:

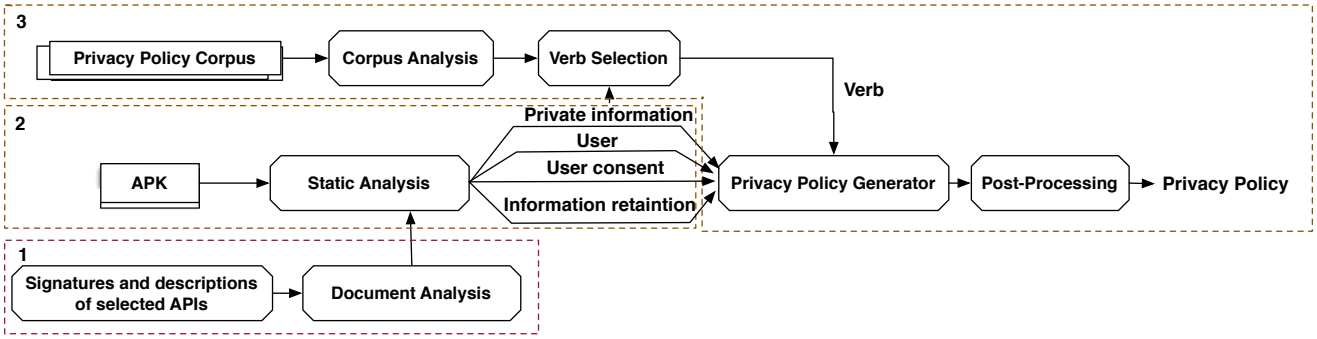

Figure 3: Overview of AutoPPG, which has three major modules in different dashed-line boxes.

(1) Document analysis module (Section 3.1). Given an API and its description from the API document, this module identifies the private information used by this API automatically by leveraging the Google Java Style [9] and employing information extraction techniques [24]. The output of this module is used by the static code analysis module to determine the private information collected by an app.

(2) Static code analysis module (Section 3.2). Given an app *without* source code and the mapping of selected APIs to the private information, this module disassembles the app's APK file, turns its statements into intermediate representation (IR), and inspects the IRs to profile the app by performing the following four steps: (1) find the private information collected by apps; (2) locate the collector of certain private information identified in (1); (3) determine whether the app asks for user consent explicitly before invoking the selected APIs; (4) identify the app's information retention strategy.

(3) Privacy policy generation module (Section 3.3, 3.4, and 3.5). Taking in an app's profile identified through static code analysis, this module aims at generating readable privacy policy. More precisely, we define a template following the instructions of writing plain English [29](Section 3.4), then select suitable verbs according to the private information (Section 3.3) to generate sentences, and finally change the order of sentences and remove duplicate sentences to improve the expressiveness (Section 3.5).

3.1 Document analysis

In this section, we detail the document analysis module for identifying the private information used by an API. We accomplish it by proposing a new algorithm (i.e., Algorithm 1), which takes the official API description, method name, and class name as input and returns the private information used by the API.

We use the API descriptions because they provide more information about an API's behaviour, such as which information the API can use or get. To extract the noun phrases referring to the collected private information, we need to locate the main verb and the object of this verb from the description. For example, function *getDeviceId()*'s description is: "*Returns the unique device ID, for example, the IMEI for GSM and the MEID or ESN for CDMA phones*", where the main verb is "Returns", and its object is "device ID".

Using APIs' descriptions as input is not enough due to the difficulty of handling all possible descriptions. For instance, given a sentence with postpositive attributive (*e.g.*, "a list of ... that ..."), the tool (i.e., Stanford parser [13]) used for analyzing the sentence will extract the object (i.e., "list"), which cannot help users understand the details of private information. Therefore, we add two kinds of information including method names and class names as the reference for determining which attributes need to be extracted.

Android framework's method names usually follow the Google java style [9] and are written in *lowerCamelCase*, which means that the first letters of all words are written in uppercase except the first word. Moreover, these method names typically contain verbs and their objects, such as **getDeviceId()**. If a method name is the verb such as *open* while its class name is the noun or noun phrases (e.g. "Camera") [9], we locate the private information from the class name. Note that Android framework's class names are typically written in *UpperCamelCase*, which means all words start with uppercase letters.

3.1.1 Sensitive APIs selection

We are interested in APIs that can get the private information. Since Susi [32] provides a list of such functions, we select 79 APIs that can obtain the following information: device ID, IP address, location (include latitude, longitude), country name, postal code, account, phone number, sim serial number, voice mail number, audio/video, installed application, visited URLs, bookmark, and cookies. We combine these APIs' signatures, which contain class names, return values, method names and parameters with their corresponding descriptions obtained from the official website, and conduct the pre-processing on them.

3.1.2 Pre-processing

As APIs' signatures and descriptions cannot be directly used as the input of our algorithm (Alg 1), we first conduct syntactic analysis on description sentence. We also extract noun phrase from method name and class name.

Syntactic analysis on API descriptions For each privacy policy, its description is a complete sentence. To identify the main verb and its corresponding object, we use Stanford parser [13], a very popular syntactic analysis tool [15, 40], to construct the sentence's syntactic structure.

Given a sentence, Stanford parser outputs its parse tree and typed dependency information. An example is shown in Fig. 4. Parse tree is the hierarchy structure, thereinto, each line represents a distinct phrase of the input sentence. The part-of-speech tags are also contained in parse tree. A part-of-speech tag is assigned to the category of words

which have the similar grammatical properties. These words usually play similar roles within the grammatical structure of sentences. Common English part-of-speech tags include NN(noun), VB(verb), PP(preposition), NP(noun phrase), VP(verb phrase), etc. Typed dependency shows the grammatical relationships between the words in a sentence. Virtual node "Root" has a root relation that points to the root word of the sentence. Other common relations between words include dobj(direct object), nsubj(nominal subject), prep_on (preposition on), etc.

Extracting noun phrase from method name. Our algorithm (*i.e.*, Algorithm 1) uses the noun phrase contained in the method name as a reference. The steps for extracting the noun phrase are presented as follows:

- **Removing verb prefixes**. The names of the majority of information retrieval functions in Susi [32] are verb phrases. They usually start with a verb prefix, such as "get" or "read", and the verb prefix is followed by the related private information. In addition, the verb prefix starts with a lowercase letter and all following words start with the uppercase letters. We remove the verb prefix and keep the noun phrase. To achieve it, we construct a verb prefix list, which contains 178 verbs, by extracting all prefixes that appear more than once in Susi's function list [32].
- **Splitting the remaining string into words**. After removing the verb prefix of the method name, we split the remaining string into distinct words by exploiting the fact that they start with uppercase letters.
- **Removing stop words**. Stop words are meaningless and they should be removed to make the extracted noun phrase more clear. We employ the stop words list provided by NLTK [2] to complete this task.

Using the method *getAllVisitedUrls* as an example to illustrate the above steps, we first remove the verb prefix "get", and then divide the remaining string into three words including "All", "Visited", and "Urls". Finally, in the last step, "All" is removed, and only "Visited Urls" is kept.

Extracting noun from class name. The fully qualified name of a class consists of the package name and the class name. For example, "android.hardware.Camera" combines package name "android.hardware" and the class name "Camera". Currently, we just extract the class name and transform it into a list of distinct words.

3.1.3 Private information extraction

Given the syntactic information of an API's description, the noun phrase from the API name, and the noun in its class name, Algorithm 1 extracts the private information used or obtained by the API. We use the API *getRunningAppProcesses()* as an example (Figure 4) to explain it.

In line 1-2 of Alogrithm 1, we get the description *desc*'s syntactic tree $desc_{tree}$ and the typed dependency relation $desc_{dept}$. The root word usually is the main verb of a sentence. In line 3, we identify the root word *root* from the typed dependency. In Figure 4, verb "Return" is extracted. Then, in line 4, we extract the direct object of the root word using function $ExtractObj$. In Figure 4, the object of verb "Return" is "list".

The object that we find in line 4 is only one word. If there are other adjective words or nouns which modify this

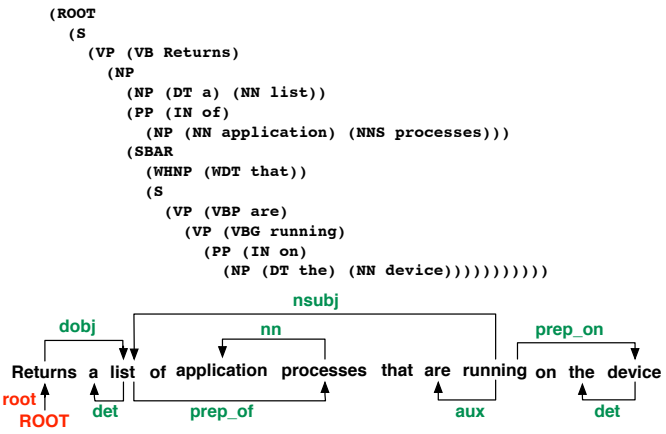

Figure 4: Syntactic analysis result, including parse tree and typed dependencies for API *getRunningAppProcesses()*'s description: *"Returns a list of application processes that are running on the device"*.

object word in typed dependency, we will also extract these adjective words or nouns and put them before the direct object *obj*. For example, for the noun phrase "device ID", if "ID" is the direct object of a verb, "device" should also be extracted and put before "ID".

Then, we need to use additional information to check if current object *obj* contains enough information. We have two kind of additional information, including noun phrase in method name, and nouns in class name. We first extract the noun phrase from the method name (line 6-7). Since some methods' names are verbs such as *start* and they do not contain any noun phrases, Algorithm 1 also extracts the nouns in the class name(line 8-10). If neither of them contains additional information, the algorithm just returns *obj* as the private information (line 26).

After getting $nameInfo$ from the method name and the class name, we calculate the semantic similarity between *obj* and $nameInfo$ in order to determine if additional postpositive attributives need to be added after current *obj*. If one of the following two conditions is met, we think that *obj* is similar to $nameInfo$ and *obj* can be directly returned. The first condition is that the semantic similarity value ($simValue$) calculated by ESA [17] is larger than the threshold (line 12-14); the second condition is that all distinct words appeared in $nameInfo$ are included in *obj* (line 16-18).

Function $FindInfo$ in line 20 is used to find the postpositive attributives of the direct object. If the subtree of the direct object *obj* contains phrases "of ..."/"from ..."/"that ..."/"for ..." in $desc_{tree}$, these phrases $info$ are intended to modify *obj* and they are also extracted.

In line 21-22, additional attribute $info$ is added after *obj* and the new *obj* is returned as the final result.

In Figure 4, since the object "list" does not represent the information "Running App Processes" obtained from method name, the final private information is "list of application processes", which combines the postpositive attributive and the object. Table 1 lists some samples of identified private information from APIs.

Semantic similarity comparison. We currently use Explicit Semantic Analysis(ESA) [17] to compute semantic relatedness between texts (line 12). Given two documents,

Input: *desc*: API description, $name_{method}$: name entity in method name, $name_{class}$: last part of class name
Output: private information used in this API

```
desc_tree = StanfordParser_Tree(desc);
desc_dept = StanfordParser_Dept(desc);
root = ExtractRoot(desc_dept);
obj = ExtractObj(desc_dept, root);
nameInfo = null;
if Exist(name_method) then
    nameInfo = name_method;
else
    nameInfo = name_class;
end
if nameInfo != null then
    simValue = Similarity(obj, nameInfo);
    if simValue > threshold then
        return obj;
    else
        ConatinName=obj.contain(nameInfo);
        if ContainName == true then
            return obj;
        else
            info=FindInfo(desc_tree, desc_dept);
            obj = obj + info;
            return obj;
        end
    end
else
    return obj;
end
```

Algorithm 1: Identifying the private information used or obtained by an API.

API	Private information
getSubAdminArea()	subadministrative area name
getAccountsByType()	accounts of a particular type
getNumberFromIntent()	phone number from an Intent
getAllVisitedUrls()	site urls

Table 1: Examples of the identified private information from APIs.

ESA uses machine learning technique to map natural language text into a weighted sequence of Wikipedia concepts. Then ESA computes the semantic relatedness of texts by comparing their vectors in the space defined by the concepts. We do not use WordNet [5] since WordNet can only calculate the similarity between distinct words. ESA can compute the relatedness between arbitrarily long texts. In Figure 4, semantic relatedness between *obj* "list" and "Running App Processes" calculated by ESA is lower than the threshold (0.5 in our system), thus *obj* cannot be regarded as the private information and more information should be added after it.

After mapping APIs to the private information, we manually group APIs that request the same kinds of private information together. For example, *getAccountsByType* and *getAccountsByTypeAndFeatures* are in the same group because they all return the list of all accounts. These API groups are used to remove the duplicate sentences (Section 3.5).

3.2 Static code analysis

Given an app, we perform the static code analysis to identify four kinds of information:

- Private information collected by this app.
- Third-party libraries used in this app and the private information collected by them.
- Conditions under which private information is collected.
- Whether the private information will be retained or not.

Our static analysis is based on Vulhunter [30] which can construct an App Property Graph (APG) for each app and store it in the Neo4j graph database [3]. APG is a combination of AST (Abstract Syntax Tree), MCG (Method Call Graph), ICFG (Inter-procedural Control Flow Graph), and SDG (System Dependency Graph).

To identify the private information collected by an app, we look for the selected 79 sensitive APIs. Note that the document analysis module has created a mapping between each API and the private information.

App can also get the private information through content providers such as "content://contacts". PScout [11] identifies 78 URI strings for Android 4.1.1. We select 8 URI strings that request the private information including contacts (3 URI strings), calendar, browser history, SMS (2 URI strings) and call log. We will examine other URI strings in future work. PScout [11] also provides 31 permissions and their related URI fields, We select 6 permissions from them and these permissions contain 615 URI fields. Since the 8 URI strings and 615 URI fields do not have the API description, we manually define the corresponding private information and "personal information" which are used as the object when generating the sentence. If query functions (e.g. *ContentResolver.query()*) and URI strings(or URI fields) are adopted together, we map the URI strings(or URI fields) to the corresponding private information.

In order to find the used third party libraries and the private information collected by them, we record the class name of the statement which called sensitive APIs or used selected content providers. If the class name is the same as one third party library's class name, we deem that the third-party library collects the private information.

3.2.1 User consent analysis

Sometimes the private information is collected with user consent (e.g., after clicking a button). To locate such execution conditions, we record the name of the method containing the statement that calls sensitive API.

In Android, there are different ways to intercept the events from users. First is event listener, when the class *View* which the listener has been registered to is triggered by the user, the callback method contained in listener is called; second is event handles, they are the callback methods used as default event handlers when developer builds a custom component from *View* [1]. If the method name is the same as an event listener's callback method or an event handler's callback method in *View* (Table 2) [1], we regard this behaviour to be executed under users' consent. Besides checking the method that contains the statement, we also perform depth first traversal in MCG (method call graph) and find all methods that call this method. If one of these methods exists in Table 2, we also deem that this behaviour will ask for user consent.

Some apps having privacy policies that notify users which information can only be used with user consent. For ex-

Event listener for view class	Event handler for view class
onClick()	onKeyDown()
onLongClick()	onKeyUp()
onFocusChange()	onTrackballEvent()
onKey()	onTouchEvent()
onTouch()	onFocusChanged()
onCreateContextMenu()	

Table 2: Callback functions for event listener and event handler in view class

```
protected void onCreate(Bundle savedInstanceState) {
 ...
      this.otherCamera.setOnClickListener(new View$OnClickListener() {
           public void onClick(View v) {
               if(Camera.getNumberOfCameras() >= 2) {
                   CameraActivity.this.mCamera.release();
                   CameraActivity.this.mCamera = Camera.open(CameraActivity.currentCameraId);
               }
           }
       });
 ...
}
```

Figure 5: App io.avocado.android: Open camera only when pressing button

ample, the location service app "com.zoemob.gpstrackin"'s class com.zoemob.familysafety.ui.bg calls *getLatitude()* function to get user location information, and this invocation is in an *onClick()* method. In its privacy policy, we find such a sentence: *"In the event of TWT DIGITAL offering a service based on the sharing of location information, we will present you with an optin for you to choose whether to participate".*

However, some apps do not declare such conditions. For example, the social app *io.avocado.android* (Fig. 5) contains an activity in which we can find a method *onClick()* calling *android.hardware.Camera.open()*. This behaviour is triggered by users. However, its privacy policy does not mention this behaviour.

3.2.2 Information retention analysis

Some private information will be not only used but also sent out through internet/SMS or saved to file/log. These specified APIs through which data could send out are called sink functions [32]. Such behaviour should be identified and declared in privacy policies to notify users. To capture such behaviour, we perform the depth first traversal from those sensitive APIs to selected sink functions based on the data dependency relation.

For example, in app *com.abukai.expenses* (shown in Fig. 6), we find that class com.abukai.expenses.Uti calls *getLongitude()* and *getLatitude()* functions, and writes the results to the file through FileOutputStream's *write()* function. However, in its privacy policy, we cannot find any statement mentioning the record of user's location information.

```
public static void a(Context arg8, String arg9, String arg10, boolean arg11, boolean arg12) {
    ...
                if(Util.d != null) {
                    v4 = Util.a(Util.d.getLongitude());
                    v3 = Util.a(Util.d.getLatitude());
                }
                Util.a(arg8, arg9, v2, v3, v4, arg10, arg11, arg12);
    ...
}
private static void a(Context arg3, String arg4, al arg5, String arg6, String arg7, String arg8,
        boolean arg9, boolean arg10) {
    ...
        FileOutputStream v1 = new FileOutputStream(String.valueOf(Util.c(arg3, arg4)) + ".DAT");
        if(arg6.length() > 0) {
            v1.write(arg6.getBytes());
        }
        if(arg7.length() > 0) {
            v1.write(arg7.getBytes());
        }
    ...
}
```

Figure 6: *com.abukai.expenses* obtains the location information and saves it into a file.

3.3 Selecting proper verbs

The verb is an important component of a sentence. After finding an app's behavior of collecting certain private information, we select a proper verb for generating the corresponding sentence. Many verbs, such as use, collect, access, read, get, take, share, retain, etc., can be used to describe the behaviours. To choose a proper verb, we take into account both the app's behaviours (Section 3.3.1) and the verb's popularity in privacy policies (Section 3.3.2).

3.3.1 Mapping behaviours to verbs

The static analysis module will provide the private information to be sent out or written to a file and the private information to be disclosed to a third-party library. They should be mapped to specific verbs. We list three cases and their candidate verbs.

Case 1: If the information collected by the app will be disclosed to a third party library, we select the verbs related to information disclosure, such as disclose, share, transfer, send, transmit, transport, etc. Note that if the information is sent out through internet, it may be saved in server and shared with the business partner. In this situation, we also select these verbs.

Case 2: If the information is retained in the file or log, we select the verbs related to information retention, such as store, save, maintain, retain, hold, keep, etc.

Case 3: If the information is neither disclosed to the third party nor saved in the file, we select verbs that do not appear in case 1 and case 2, such as collect, use, process, access, etc.

By analyzing the privacy policies of 500 apps fetched from Google Play, we find out the verbs that have been often used in privacy policies. Table 3 lists some candidate verbs for different cases.

Private Info	Case 1	Case 2	Case 3
location	share	retain	use, collect, request
video	disclose, share	store	use, take
device ID	provide	store	use, access, collect
phone number	provide	record	collect, use
site urls	disclose	log	use, collect

Table 3: Private information and their corresponding verbs in different cases

Grouping noun phrases Since different noun phrases may point to the same object, we group them together to find the related verbs. More precisely, we use ESA [17] to compute the similarity between noun phrases extracted from privacy policies with all private information extracted from API descriptions. Currently, we set the threshold value to 0.5. If the semantic similarity between one noun phrase and the private information is larger than the threshold, we regard them as the same thing and combine their corresponding verbs together.

Private information	Synonyms
location	country location, device location, location detail, device geolocation, etc.
phone number	phone number string, phone book, verified phone number, cell phone number, mobile phone contact list, etc.
ip	ipv4, visitor ip address, commenter ip, etc.

Table 4: Private information and their synonyms

3.3.2 Selecting proper verbs for private information

Once the collected private information and the behaviour case are decided, multiple candidate verbs can be used to generate sentences. We propose a method to select the best verb by analysing the privacy policy corpus in order to make the generated sentences similar to manually written ones.

According to the Bayes rule, the co-occurrence probability of the private information and the verb [41] is as follows:

$$\begin{aligned} P(Sent|n,v) =& \frac{P(n,v|Sent)P(Sent)}{P(n,v)} \\ =& \frac{P(n|Sent)P(v|Sent)P(Sent)}{P(n)P(v)} \end{aligned} \quad (1)$$

In the above equation, n refers to the private information, $Sent$ denotes all sentences contained in privacy policies, and v is the verb. This probability is proportional to $P(n|Sent)*P(v|n)$ [41]. For each kind of private information (object, n), we count the number of sentences that contain the related noun phrases to calculate the possibility $P(n|Sent)$. Then we compute each verb's appearance possibility $P(v|n)$ in these sentences. Finally, we select the verb which has the highest co-occurs possibility as the best one.

3.4 Privacy policy generator

Sentence generation. In section 3.1, we map each API to the corresponding private information n. We find the collected private information and the collector of such private information in section 3.2. And in section 3.3, the best verb v of each private information n is found. Note that the collector and the private information n serve as the subject and the object of the generated sentence for privacy policy, respectively.

The generated sentence adopts plain English [29], which allows readers to get the message easily. Plain English has an average sentence length of 15-20 words, and it prefers active verbs. Therefore, we define the structure of each generated sentence as:

$$sentence = subject\ verb\ object\ [condition]$$

This structure contains four parts.

- The subject of the sentence is the private information collector. If it refers to a third-party library, we use the library's name as the subject; otherwise, we use "We" as the subject of sentence.
- Object is the collected private information. This is determined by the mapping from sensitive APIs to the private information (Section 3.2).
- The verb gotten from privacy policy corpus is used as the verb of the sentence.
- Condition is an optional element. If the API is triggered by button clicking, we add "with your consent" after the sentence; otherwise, this part is removed.

Additional Sentence. If an app does not collect or use any personal information or third party libraries, we cannot generate any sentence to describe its behaviour. In this case, providing an empty privacy policy is improper and we need to add the additional sentences according to the following options: (1) If the app itself does not collect any personal information, we add the sentence "We do not collect any of your personal information" to the privacy policy; (2) If the app does not use any third party libraries, we add the sentence "We do not use any third-party libraries in this app". If third party-libraries can be found, we also add a sentence "The following third party libraries are used:" before sentences that related to these third party libraries.

3.5 Post-Process: Removing duplicate sentences and rearranging sentences

The generated privacy policies may contain duplicate sentences in random order. To improve the readability of these privacy policies, we perform two additional process: removing duplicate sentences and changing sentences order.

Removing duplicate sentences. Duplicate sentences refer to multiple sentences having the same meaning in one privacy policy. It may appear due to two reasons: (1) Direct duplicates. An app calls different APIs that get the same kind of information. For example, there are three APIs, including *getAccounts()*, *getAccountsByType()*, and *getAccountsByTypeAndFeatures()*, for obtaining the account information. If they appear in one app, AutoPPG may generate three sentences. (2) Hidden duplicates. The information that one API gets can cover that obtained through another API. For example, an app invokes both *getLastKnownLocation()* and *getLatitude()*. The former function gets the location information, while the latter obtains the latitude. AutoPPG will generate two sentences for them. However, since the location information is more general than the latitude, we can combine the two sentences into one.

To remove duplicates in the first case, APIs requesting the same kind of information should be grouped together. When generating sentences, if multiple APIs in the same group are called **by the same collector**, only one API is used to generate the sentence. We also group those content provider's URI strings and URI fields to achieve the same purpose.

To remove the duplicates in the second case, we build a tree structure for personal information. Note that if the information on the parent node is collected, information collected in its sub-tree can be covered. We use a personal information ontology [18] [20] to build up such model. Ontology is a formal representation of a body of knowledges within a given domain [10]. The personal information ontology [18] covers all personal information of a person, and all personal information is organized in hierarchy structure (Fig. 7). We put all APIs and URI(fields) into corresponding classes and properties in this ontology. For example, *getLastKnownLocation()* is put into class node "location", and *getLatitude()* is put into property node"latitude". Then we perform a depth first search on the ontology, if one API belonging to the parent node is called and another API belonging to the child node is also invoked, we remove the sentences generated by the child node and add its private information into the parent node's sentence.

Rearranging sentences. After generating a number of sentences, we rearrange them according to the importance degree of the corresponding private information. In [16], Felt et al. surveyed 3,115 smart-phone users and asked them to rate how upset they can be if the given risks occurred. According to the "Upset Rate" for different kinds of risks, we get a risk rank for different kinds of personal information (Table 5). Based on this rank, we change the appearance order of sentences before writing them to file. For the infor-

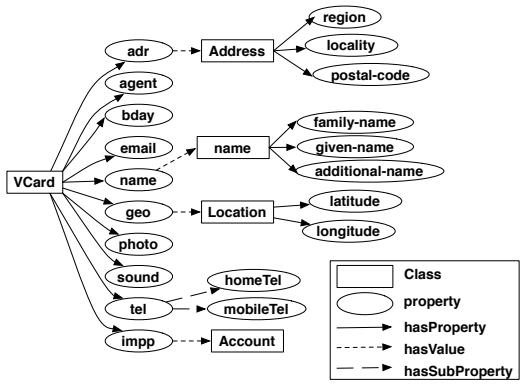

Figure 7: Personal information ontology getting from *http://www.w3.org/TR/vcard-rdf/*

mation (*e.g.*, cookie) which does not appear in [16], we put them at the end of the privacy.

#	Private Information	#	Private Information
1	contact	7	audio
2	SMS	8	camera
3	call log	9	location
4	browser history	10	account
5	calendar	11	app list
6	device ID		

Table 5: Private information risk rank list

Fig. 8 shows a sample privacy policy generated by AutoPPG. Note that, since we do not find any information that is collected by app and disclosed to third-party libraries, all sentences in Fig. 8 do not use verbs related to information disclosure as their main verbs.

```
Information we would collect/save:
we use contacts;
we use SMS;
we use call log;
we use calendar;
we use unique device id;
we use phone number for line 1;
we use location area code;
we use localized country name;
we would record latitude;
we collect latitude with your consent (agree and click some button);
we collect longitude with your consent (agree and click some
button);
we collect ip address;

Third Party Libraries and the information collected by them:
The following third party libraries are used: Google Analytics,
Fortumo, Flurry Analytics.
Some of your personal information may be collected by them:
Google Analytics collect longitude;
Google Analytics collect latitude;
Fortumo use unique subscriber id;
Fortumo use phone number for line 1;
Flurry Analytics use unique device id;
Flurry Analytics use name of the provider;
Flurry Analytics use location;
```

Figure 8: Example: The document AutoPPG generate

4. EXPERIMENT

We conduct extensive experiments to answer the following research questions.

- **RQ1-Correctness:** *How is the correctness of the document analysis module?* The purpose of this research question is to check the correctness of AutoPPG's document analysis module.
- **RQ2-Expressiveness:** *How is the expressiveness of the generated privacy policies?* In this part, we will explore whether the generated privacy policies are easy to read and understand for users.
- **RQ3-Adequacy:** *Do the generated privacy policies satisfy the requirement?* For this research question, we aim to verify whether the generated privacy policies meet the requirements, especially in telling users all collected private information.

In the following subsections, we first introduce the data set and then detail the experimental results.

4.1 Data set

We download 7,181 apps from Google play, all of which provide privacy policies. We randomly select 500 apps containing privacy policies in English from them as our dataset.

Since some experiments, which require human intervention, cannot been done in large scale, we randomly select 12 apps' privacy policies for such experiments. They include 8 manually written and 4 template-based privacy policies. The names of these apps are shown in Table 6.

4.2 Answer to RQ1: The correctness of the document analysis module

The precision of mapping APIs to their collected information. Instead of using the 79 APIs mentioned in Section 3.1.1, we randomly select another 67 source functions that collect certain information from Susi [32], because we aim to evaluate the AutoPPG's accuracy of mapping APIs to their collected information through text analysis. We use these APIs' signatures and descriptions as the input of our document analysis module and manually check the output.

We found that only 3 out of 67 APIs have incorrect results. For example, the description of the API *Cursor.getColumnNames()* is: "*Returns a string array holding the names of all of the columns in the result set in the order in which they were listed in the result*". AutoPPG gets the object "a string array" of the verb "Returns", but misses "names of all of the columns". It is due to the fact that in the output of Stanford parser "names of all of the columns" is not in the subtree of object "a string array" and our algorithm only searches attributes in the subtree of the object.

The precision of our document analysis module is computed by the following formula:

$$precision = \frac{True\ Positive}{Total} = 95.5\%,$$

where $True\ Positive$ stands for the number of APIs that document analysis module can correctly extract the related information while $Total$ means the total number of APIs used in this experiment.

The effectiveness of duplicate sentences removal. To illustrate the necessity of removing duplicate sentences, we count the number of generated sentences in each app before and after removing the duplicates. The result of CDF is shown in Fig. 9. The blue curve is on the left of red curve, meaning that many apps have been affected by removing duplicate sentences.

Num	App	Num	App
1	air.bftv.larryABCs	7	com.gamevil.spiritstones.global.free
2	air.com.arpamedia.candyprincessmakeover	8	com.godaddy.mobile.android
3	air.com.gamesys.mobile.slots.jpj	9	com.goodreads
4	air.com.kitchenscramble.goo	10	com.newzbid.newzapp
5	air.com.playtika.slotomania	11	com.roadwarrior.android
6	com.crimsonpine.stayinline	12	llc.auroraappdesign.votd

Table 6: The apps that we use for manual check

Without deduplication, AutoPPG generates 7,801 sentences for 500 apps in total. After AutoPPG removes duplicate sentences, only 6,039 sentences are left. In other words, 22.5% sentences have been eliminated.

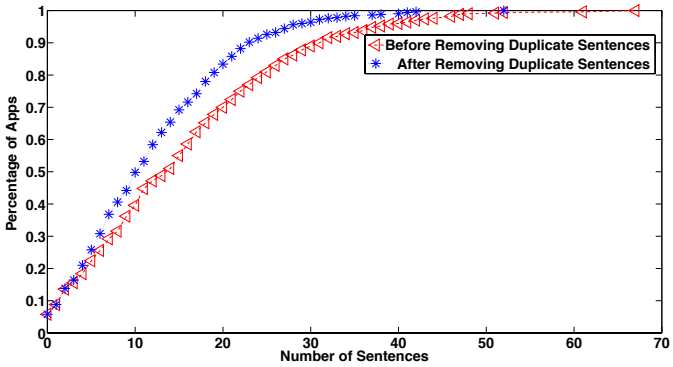

Figure 9: Number of sentences before and after removing duplicate sentences

4.3 Answer to RQ2: The expressiveness of the generated privacy policies

We measure the expressiveness of the generated privacy policies from two aspects: readability and understandability.

We employ crowdsourcing to compare the readability and understandability between privacy policies generated by AutoPPG and the existing ones. More precisely, we randomly select 12 apps as examples, and put each app's existing privacy policy and the generated one into the questionnaire. Then, we publish these 12 questionnaires on **Amazon Turk**, and ask workers in crowdsourcing to read them.

After reading a privacy policy, each worker will answer two questions. Q1 is "Is this privacy policy easy to read?". This question is proposed to compare the readability of the existing privacy policy and that of the generated privacy policy. We provide four answers to choose, including "very difficult", "difficult", "easy", and "very easy".

Q2 is: "Do you think whether an app with such behaviour violates users' privacy?" This question is proposed to evaluate which description is more understandable. We also provide four answers to choose, including "Absolutely NO", "NO", "YES", and "Absolutely YES". Since apps' existing privacy policies contain some irrelevant information, we remove them and only reserve the sentences related to collecting and using private information in the questionnaires.

After publishing them on Amazon Turk for 3 days, we finally got 189 responses from 66 workers, and each privacy policy has been read by at least 10 workers.

The distribution of all responses to Q1 is shown in Fig. 10. We can find that for the generated privacy policies, the proportion of readers choosing "very easy" is larger than that for the existing privacy policies, and the proportion of readers selecting "very difficult" is less than that for the existing privacy policies. Note that, although the number of workers who select "easy" for the existing privacy policies is larger than that for the generated privacy policies, more workers select "very easy" for the generated privacy policies.

Fig. 11 shows the Q2's answer distribution. We can see that after reading generated privacy policies, most of workers choose "YES" and "Absolutely YES", indicating that the generated privacy policies are more understandable that existing ones.

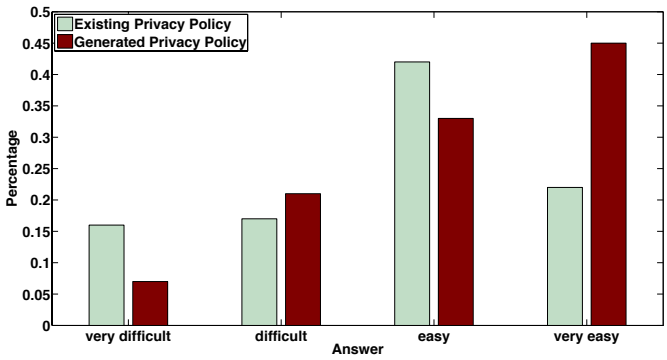

Figure 10: Readability comparison between existing privacy policy and new generated privacy policy: Is this privacy policy easy to read?

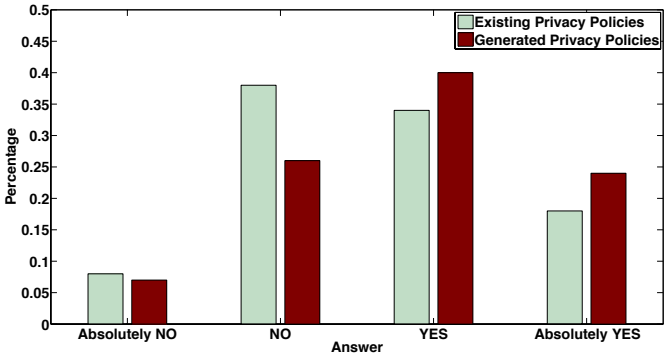

Figure 11: Understandability comparison between existing privacy policy and new generated privacy policy: Do you think an app with such behaviour violate user privacy?

4.4 Answer to RQ3: the adequacy

If a developer uses many different APIs or various third-party libraries, he or she may miss some information when

writing privacy policies manually. In contrast, AutoPPG can achieve a low false negative rate because AutoPPG conducts static code analysis. In this experiment, we use 12 randomly selected apps and compare the coverage of the generated privacy policies on those collected private information and that of the existing privacy policies.

Information selection. We select 9 kinds of private information, including location, account, video, audio, device ID, IP address, app list, cookie, and phone number. We also count the number of third-party libraries mentioned in the privacy policies.

Table 7 shows the comparison result. We use "T" to represent template-based privacy policies , "N" to stand for new generated policies, and "O" to indicate exist privacy policy files. From Table 7, we can see that the total number of the generated privacy policies that contain any kind of private information is larger than the number of existing privacy policies that contain the corresponding information. Moreover, the number of third-party libraries covered by the generated privacy policies is also larger than that in existing privacy policies.

As an example for app 7, the existing privacy policy only tells users that social account and related information can be used, while our generated privacy policy can inform users that the application list, contacts, device ID, phone number, and account will be used. Moreover, the app 7's existing privacy policy does not contain the names of third-party libraries, while AutoPPG identifies and lists 7 third-party libraries used by this app.

5. THREATS TO VALIDITY

In this paper, we develop a tool called AutoPPG to automatically construct correct and readable descriptions to facilitate the generation of privacy policy for Android apps. We list potential threats to validity in our work as follows.

Design target. Privacy policy is a complex document and AutoPPG can only generate statements related to apps' behaviours. Some other information, such as "how to contact the developer" and "how can user ask the developer to delete their personal information", still needs the developers to write by themselves. Static analysis can find an app's privacy-related behaviours in code, but it cannot discover the purpose of these behaviours. Therefore, the purpose of each behaviour needs to be added by developer after applying AutoPPG to the app.

System design. The static analysis module of our system uses Epicc [28] to find the target of the intent. However, advanced tools such as Iccta [22] can also be used to find the communication between components. When discovering third party libraries, more libraries' class names can be added to the white list to improve our system's coverage.

When checking parameters of the query function of the content provider, our system cannot handle the complex string operations such as "append", "split". This issue may lead to false negatives [37], and we will tackle it in future work.

When finding the behaviours triggered by users, AutoPPG only checks method names. This may cause false positives, and additional check will be added in future work, such as dynamic checking.

Experimental result. In our experiment, we selected 76 sensitive APIs, 8 URI strings, and 615 URI fields as sources. In future work, more APIs will be examined.

When comparing the readability, we only selected 12 apps as samples since removing useless sentences needs much manpower and only 66 workers answered the questionnaires. We will investigate more samples and involve more readers in future work. Since Amazon Turk does not allow us to get the personal information of workers, such as "location", "age", we cannot show the demographic information.

6. RELATED WORK

AutoPPG performs static analysis on android apps to generate the privacy policies. In this section, we introduce some related works on privacy policy, privacy policy generator, and software artifacts generation.

6.1 Privacy policy

Since privacy policies are written in natural language, some studies investigated how to parse privacy policies and mine information from them. Breaux et al. [12] present a methodology for directly extracting access rights and obligations from the US Health Insurance Portability and Accountability Act (HIPAA) Privacy Rule. To automate the process, semantic patterns are employed to process the access control policy [39] [38].

Some studies used machine learning techniques to analyse the features of privacy policies. Costante et al. use machine learning algorithm to analyse the completeness of privacy policies [14]. Zimmeck et al. combine crowdsourcing and machine learning technique to check essential policy terms in privacy policies [42]. Liu et al. adopt HMM to perform the automatic segment alignment for policies [23] [31].

All these works are based on existing privacy policies and make no effort to actively generate new privacy policies.

6.2 Privacy policy generator

Although Privacy Informer [26] was proposed to automatically generate privacy policies, it can only analyse the source code of mobile apps created through App Inventor, but it cannot generate privacy policies for normal apps due to the lack of source code.

PAGE [35] is an Eclipse plug-in that enables developers to create privacy policies during the developing process. However, PAGE is similar to those online privacy policy generators since it does not analyse apps' source code and it requires developers to answer a list of questions such as "What type of data do you collect" and "Do you share data with others".

AppProfiler is another similar work [34]. It detects privacy-related behaviors in apps and explains them to end users [34]. However, AppProfiler is based on the manually created knowledge base which maps the privacy-relevant APIs to appropriate behaviours. Instead, AutoPPG can automatically map APIs to their collected private information by using information extraction techniques. Moreover, AppProfiler can only identify system calls, method name, and class name, but it cannot perform the data flow analysis and cannot find which information will be retained.

6.3 Software artifact generation

Privacy policy is a category of software artifact, thus the previous studies related to other software artifacts (*e.g.*, test case and test code) provide the inspiration for our work. Rayadurgam et al. proposed a method to generate the test cases according to structure coverage criteria. The de-

Info	APP 1		APP 2		APP 3		APP 4_T		APP 5		APP 6		APP 7		APP 8		APP 9		APP10_T		APP11_T		APP12_T		Total	
	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O
Device ID	*				*		*	o	*	o			*		*	o	*								7	3
IP address			*		*		*	o		o							*								4	2
location	*	o	*		*		*		*	o							*		o		*	o			7	4
account	*		*		*	o				o			*	o							*				5	3
video/camera	*	o	*		*		*		*								*								6	1
audio	*	o	*		*		*		*																5	1
app list			*				*		*		*		*		*										6	0
cookie	*			o		o		o		o		o			*			o	*	o		o		o	2	9
phone number								o					*												1	1
Used Lib Num	0	0	3	2	4	0	7	1	3	0	3	3	7	0	0	0	1	0	0	0	1	1	4	4	33	11

Table 7: Comparison result of coverage of generated privacy policies and existing privacy policies: "N" means new generated, "O" means exist old privacy policy, $_T$ means template-based privacy policy

signed model checker can produce complete test sequences that provide a pre-defined coverage of any software artifact [33]. Javed et al. proposed a new method to generate the test cases automatically. This method used the model transformation technique to generate these test cases from a platform-independent model of the system [21]. Harman et al. proposed a mutation-based test data generation approach that combines dynamic symbolic execution and search-based software testing. This hybrid generation method gets the better performance than state-of-the-art mutation-based test data generation approaches [19]. Matthew et al. proposed a tool to generate test code in model-driven systems. Moreover, they quantified the cost of the test code generation versus application code and found that the artifact templates for producing test code are simpler than those used for application code [36].

7. CONCLUSION

We propose and develop a novel system named *AutoPPG* to automatically construct correct and readable descriptions to facilitate the generation of privacy policy for Android applications (i.e., apps). AutoPPG can identify the private information collected or used by an API from its description, name, and the class name. It can also discover an app's behaviors related to users' private information by conducting static code analysis, and generate correct and accessible sentences for describing these behaviors. The experimental results using real apps and crowdsourcing demonstrate the correctness of AutoPPG's document analysis module, the expressiveness, and the adequacy of the generated privacy policies. In future work, we will further improve the performance of the document analysis module and the static code analysis module. Moreover, we plan to examine more apps and involve more persons in the evaluation of AutoPPG.

8. 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 (G-UA3X), and the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province (2013GDDSIPL-04).

9. REFERENCES

[1] Android input event. http://goo.gl/enQtDI.
[2] Natural language toolkit. http://www.nltk.org/.
[3] Neo4j graph database. http://neo4j.com/.
[4] Privacy policies for android apps. https://termsfeed.com.
[5] Wordnet. http://wordnet.princeton.edu/.
[6] Mobile privacy disclosures: Building trust through transparency. Technical report, FTC Staff Report, 2013.
[7] The need for privacy policies in mobile apps − an overview. http://goo.gl/7AB2aB, 2013.
[8] Path social networking app settles ftc charges it deceived consumers and improperly collected personal information from users' mobile address books. https://goo.gl/Z31BAU, 2013.
[9] Google java style. https://goo.gl/1RtxN1, 2015.
[10] Ontology structure. http://goo.gl/sBGgzZ, 2015.
[11] K. Au, Y. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proc. ACM CCS*, 2012.
[12] T. Breaux and A. Anton. Analyzing regulatory rules for privacy and security requirements. *IEEE Trans. on Software Eng.*, 2008.
[13] D. Cer, M. Marneffe, D. Jurafsky, and C. Manning. Parsing to stanford dependencies: Trade-offs between speed and accuracy. In *Proc. LREC*, 2010.
[14] E. Costante, Y. Sun, M. Petkovic, and J. Hartog. A machine learning solution to assess privacy policy completeness. In *Proc. ACM WPES*, 2012.
[15] J. Cowie and W. Lehnert. Information extraction. *Communications of the ACM*, 39(1), 1996.
[16] A. Felt, S. Egelman, and D. Wagner. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *Proc. ACM SPSM*, 2012.
[17] E. Gabrilovich and S. Markovitch. Computing semantic relatedness using wikipedia-based explicit semantic analysis. In *Proc. IJCAI*, 2007.
[18] H. Halpin, B. Suda, and N. Walsh. An ontology for vcard. http://www.w3.org/2006/vcard/, 2015.
[19] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proc. ESEC/FSE*, 2011.
[20] R. Iannella and S. Identity. vcard ontology - for describing people and organizations. http://www.w3.org/TR/vcard-rdf/, 2015.
[21] A. Javed, P. Strooper, and G. Watson. Automated generation of test cases using model-driven architecture. In *Proc. AST*, 2007.

[22] 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.

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

[24] C. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

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

[26] D. Miao and L. Kagal. Privacy informer: An automatic privacy description generator for mobile apps. In *Proc. ASE PASSAT*, 2014.

[27] M. Nauman, S. Khan, and X. Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proc. ACM ASIACCS*, 2010.

[28] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proc. USENIX Security*, 2013.

[29] Plain English Campaign. How to write in plain english. http://goo.gl/qwq2Sh.

[30] C. Qian, X. Luo, Y. Le, and G. Gu. Vulhunter: toward discovering vulnerabilities in android applications. *IEEE Micro*, 2015.

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

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

[33] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. ECBS*, 2001.

[34] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proc. CODASPY*, 2013.

[35] M. Rowan and J. Dehlinger. Encouraging privacy by design concepts with privacy policy auto-generation in eclipse (page). In *Proc. ETX*, 2014.

[36] M. Rutherford and A. Wolf. A case for test-code generation in model-driven systems. In *Proc. GPCE*, 2003.

[37] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek. Information flows as a permission mechanism. In *Proc. ASE*, 2014.

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

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

[40] Y. Xu, M.-Y. Kim, K. Quinn, R. Goebel, and D. Barbosa. Open information extraction with tree kernels. In *HLT-NAACL*, 2013.

[41] Y. Yang, C. L. Teo, H. Daumé III, and Y. Aloimonos. Corpus-guided sentence generation of natural images. In *Proc. EMNLP*, 2011.

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