Toward Automatically Generating Privacy Policy for Android Apps

Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang

***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 understand the source code well as it could have been written by others (e.g., outsourcing), or the author does not know the internal working of third-party libraries used. 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 static code analysis to characterize its behaviors related to users' personal 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; 2) the privacy policies constructed by AutoPPG usually reveal more operations related to users' personal information than existing privacy policies; and 3) most developers, who reply us, would like to use AutoPPG to facilitate them.**

***Index Terms*— Mobile applications, natural language processing, privacy policy, static code analysis.**

I. Introduction

AS SMARTPHONES have become an indispensable part of our daily lives, users are increasingly concerned about the privacy issues of the personal information collected by apps. Although the Android system lists 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' personal information, due to the lack of precise and accessible descriptions [1], [2]. Alternatively, developers can provide privacy policies to their apps for informing users how their information will be collected, used, and disclosed [3]. 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 [4]. Other major countries or areas (e.g., EU,etc.) have privacy laws for requiring developers to add privacy policies [5].

Failing to provide a correct privacy policy may result in a fine. For instance, the social networking app "Path" was fined $800,000 by FTC in 2013 because it collected and stored users' personal information *without* declaring the behaviors in its privacy policy [6]. As another example, FTC filled a complaint against a popular app named "Brightest Flashlight Free" in 2014 because its privacy policy does not correctly reflect how it uses personal data [7].

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 automatically generating privacy policies cannot solve these issues because they rely on human intervention, such as answering questions like "what personal information do you collect?", and few of them analyze code. It is worth noting that the tool Privacy Informer [8] could only generate privacy policies for apps created by App Inventor [9] 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 non-trivial to develop AutoPPG because of several challenging issues. First, how to automatically map APIs to personal information? Note that other similar studies (e.g., AppProfiler [2]) did it manually. By exploiting the Google Java style followed by the Android framework [10] and information extraction techniques [11], we propose a new algorithm (i.e., Algorithm 1 in Section III-A) to address this issue.

Second, how to profile an app's behaviors related to personal information through static analysis? By leveraging the app property graph (APG) [12], we design various graph traversals to answer questions like (Section III-B): does the app collect personal information? If yes, does the host app or any third-party library collect such information? Does the app notify users before collecting the information? Will the app store the information locally or send it to a remote server?

Manuscript received March 16, 2016; revised September 6, 2016 and November 19, 2016; accepted November 21, 2016. Date of publication December 13, 2016; date of current version January 25, 2017. This work was supported in part by Hong Kong GRF under Grant PolyU 5389/13E, in part by the National Natural Science Foundation of China under Grant 61202396 and Grant 61602258, in part by the HKPolyU Research under Grant G-UA3X and Grant G-YBJX, in part by the Shenzhen City Science and Technology R&D Fund under Grant JCYJ20150630115257892, and in part by the China Postdoctoral Science Foundation under Grant 2015M582663. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Mauro Barni. *(Corresponding author: Xiapu Luo.)*

L. Yu, T. Zhang, X. Luo, and L. Xue are with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong (e-mail: cslyu@comp.polyu.edu.hk; cstzhang@comp.polyu.edu.hk; csxluo@comp.polyu.edu.hk; cslxue@comp.polyu.edu.hk).

H. Chang is with the Law and Technology Centre, The University of Hong Kong, Hong Kong (e-mail: hcychang@hku.hk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2016.2639339

Lastly, how to construct correct and readable descriptions? To address this issue, we propose several new approaches based on NLP techniques [13], including, selecting proper verbs, defining clear and simple sentence structure following the guidelines of plain English [14], removing sentences with the same meanings, and rearranging sentences according to the level of importance (Section III-C and Section III-E). We validate the generated descriptions through crowdsourcing-based experiments (Section IV-C).

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 apps. 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 in developing AutoPPG, including automatically mapping APIs to personal information, profiling an app's behaviors related to personal information, and constructing correct and readable descriptions. These techniques can also be applied to solve other research problems, such as malware detection.
3) We implement AutoPPG and perform careful experiments with real apps and crowdsourcing to evaluate its performance. The experimental results show that AutoPPG can construct correct and easy-to-understand descriptions for privacy policy. Actually, the privacy policies resulted from AutoPPG usually reveal more apps' behaviors related to users' personal information than existing privacy policies. Moreover, most developers, who reply us, would like to use AutoPPG to facilitate them.

The rest of this paper is structured as follows. Section II presents the background. Section III details AutoPPG and Section IV describes the experimental results, respectively. Section V discusses the limitation of AutoPPG and future improvement. After introducing the related work in Section VI, we conclude the paper in Section VII.

II. Background

A. Privacy Policy

According to [5], a privacy policy may contain five kinds of information: (1) contact and identity information; (2) personal information to be collected and used; (3) the reasons why the data is needed; (4) third parties to whom the information will be disclosed; (5) users' rights. Since (1), (3) and (5) cannot be identified by analyzing an app, AutoPPG focuses on generating statements for (2) and (4).

As an example, Fig. 1 shows a 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. 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. Such information belongs to (2) and (4), respectively, and AutoPPG can create them.

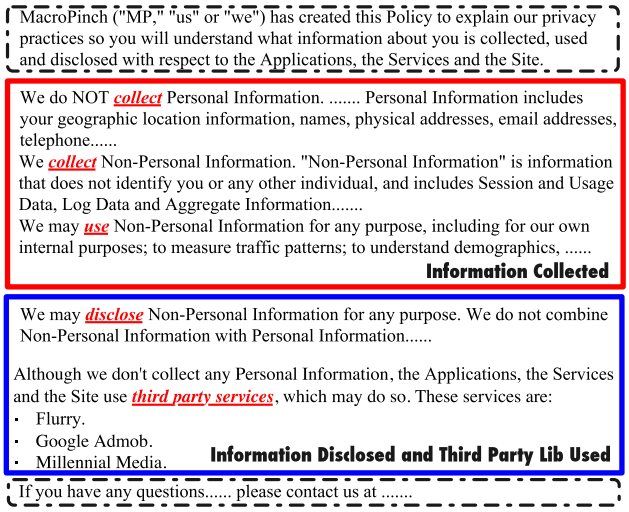

Fig. 1. Sample of privacy policy (com.macropinch.swan).

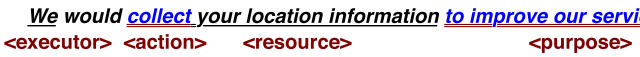

Fig. 2. The structure of a general sentence in privacy policy.

B. The Sentence Structure

A general sentence in privacy policy contains three key parts, including executor, action, and resource. Other parts such as condition (*e.g.*, when this action happens), purpose (*e.g.*, why the information is needed), and target (*e.g.*, whom the information will be sent 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 a third-party library, this information will be disclosed to the 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".

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

III. AutoPPG

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

(1) Document analysis module (Section III-A): Given an API and its description from the corresponding API document, this module identifies the personal information used by the API automatically by leveraging the Google Java Style [10] and employing information extraction techniques [11]. The output

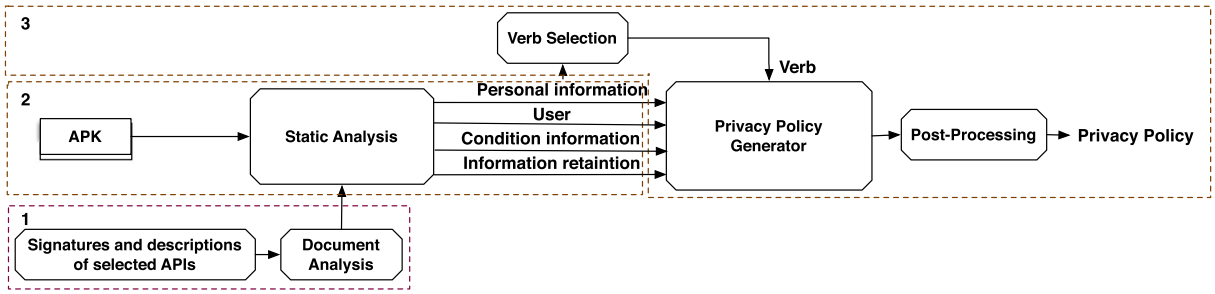

Fig. 3. Overview of AutoPPG, which has three major modules in different dashed-line boxes.

of this module is used by the static code analysis module to determine the personal information collected by an app.

(2) Static code analysis module (Section III-B): Given an app *without* source code and the mapping of selected APIs to the personal information from the document analysis module, 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) finding the personal information collected by apps; (2) locating the collector of the personal information identified in (1); (3) determining whether or not the app asks for users' consent explicitly before invoking the selected APIs; (4) identifying the app's information retention strategy.

(3) Privacy policy generation module (Section III-C, III-D, and III-E): Taking in an app's profile identified through static code analysis, this module aims at generating readable privacy policy. More precisely, we define a sentence template following the guidelines of plain English [14] (Section III-D), and then select suitable verbs according to the personal information for generating sentences (Section III-C). Moreover, we change the order of sentences and remove the duplicate sentences to improve the readability (Section III-E).

A. Document Analysis

This section details the document analysis module for identifying the personal information used by an API. Our algorithm (i.e., Algorithm 1) takes the official API description, method name, and class name as input, and returns the personal information used by the API.

We process the API descriptions because they provide more information about an API's behaviours, such as the information accessed by the API. To extract the noun phrases referring to the collected personal 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 may not be enough due to the difficulty of handling all descriptions. For instance, given a sentence with postpositive attributive (*e.g.*, "a list of …that …"), the tool (i.e., Stanford parser [15]) used for analyzing the sentence will extract the object (i.e., "list"), which cannot help users understand the details of personal information. Therefore, we leverage two additional kinds of information, including method names and class names, as the reference to determine the postpositive attributes that need to be extracted.

Android framework's method names usually follow the Google Java style [10] and are written in *lowerCamelCase*, meaning 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()*. In this case, the information contained in method name can be used as a reference. However, if the method name is a verb (e.g., the *open(int cameraId)* method in the *android.hardware.Camera* class), we cannot extract private information from its method name. Since the class name may be a noun or noun phrase (e.g., *Camera* in this example) [10], we locate the personal information from the class name as a reference. Note that Android framework's class names are typically written in *UpperCamelCase*, meaning that all words start with uppercase letters.

In the remaining of this section, we first introduce the sensitive APIs selected as the input of the document analysis module. Then, we detail the document analysis module's steps, including: 1) Pre-processing step, where we do syntactic analysis on the API description, and extract personal information from the method name and class name. 2) Personal information extraction step, where we leverage the API description's syntactic information, the information extracted from method name and class name if any to determine the personal information to be accessed by the API.

1) Sensitive APIs Selection: We are interested in APIs that can get the personal information. Since Susi [16] 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, audio/video, installed application list, visited URLs, browser bookmarks, and cookies. We combine these APIs' signatures with their official descriptions, and conduct the pre-processing on them. The signature of an API contains class name, type of return value, method name, and types of parameters of the API.

It is worth noting that apps can also obtain the personal information through content providers (*e.g.*, "content://contacts"). PScout [17] identifies 78 URI strings for Android 4.1.1. We select 8 URI strings that request the personal information, including contacts (3

URI strings), calendar, browser history, SMS (2 URI strings), and call log. Other URI strings are ignored since their information is not considered by AutoPPG (e.g.,"content://com.example.documents" is related to `MANAGE_DOCUMENTS` permission). PScout [17] lists 31 permissions and their URI fields. We select 6 permissions from them, and these permissions contain 160 URI fields. Other permissions and their related URI fields are ignored because they are not relevant to sensitive personal information (e.g., the URI fields `<android.provider.UserDictionary: android.net.Uri CONTENT_URI>` under the `READ_USER_DICTIONARY` permission).

2) Pre-Processing: We first extract personal information if any from an API's method name and class name. Then, we conduct syntactic analysis on the API's description. Finally, our algorithm (i.e., Algorithm 1) automatically determines the personal information accessed by the API.

a) Extracting Personal Information From a Method Name: We first remove the verb prefix in the method name because the names of most information retrieval functions are verb phrases. They usually start with a verb prefix, such as "get" or "read", and the verb prefix is followed by the corresponding personal information. Moreover, the verb prefix starts with a lowercase letter and all following words start with the uppercase letters. We construct a verb prefix list, which contains 178 verbs, by extracting all prefixes that appear more than once in Susi's function list [16].

Based on the verb prefix list, we remove the method name's verb prefix and split the remaining noun phrase into distinct words by exploiting the fact that they start with uppercase letters. Moreover, we remove the stop words according to the stop word list provided by NLTK [18], because stop words are meaningless and removing them can make the extracted noun phrase more clear. 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, "All" is removed, and only "Visited Urls" is kept.

b) Extracting Personal Information 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 turn it into a list of distinct words.

c) Syntactic Analysis on API Descriptions: Given a selected API's description, to identify the main verb and its corresponding object, we use Stanford parser [15], a very popular syntactic analysis tool [19], [20], to construct the sentence's syntactic structure. Stanford parser outputs the sentence's phrase structure tree and typed dependency information.

An example is shown in Fig. 4. The phrase structure tree has the hierarchy structure, where each line represents a distinct phrase of the input sentence. The part-of-speech (PoS) tags are also contained in the phrase structure tree. A PoS tag is assigned to the category of words which have the similar grammatical properties. These words usually play

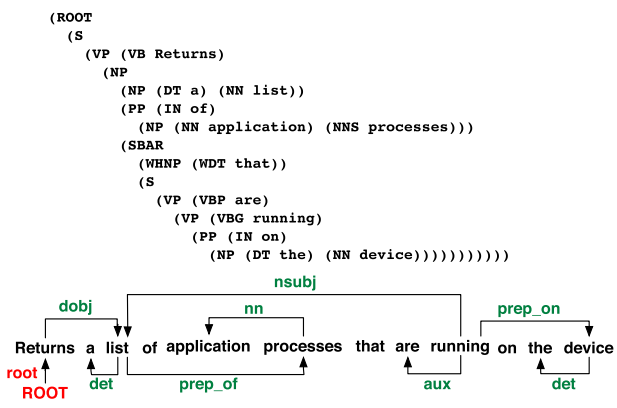

Fig. 4. The phrase structure tree and the typed dependencies of the API *getRunningAppProcesses()*'s description: *"Returns a list of application processes that are running on the device"*.

similar roles within the grammatical structure of sentences. Common English PoS 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.

3) Personal Information Extraction: Given the syntactic information of an API's description, the personal information extracted from the method name and the class name, Algorithm 1 determines the personal information obtained by the API. We use the API *getRunningAppProcesses()* as an example (Fig. 4) to explain it.

The pre-processing step provides the description sentence's (i.e., $desc$) phrase structure tree $desc_{tree}$ and typed dependency information (i.e., $desc_{dept}$) (line 1-2 in Algorithm 1).

Using the typed dependency information, we can identify the root word $root$ from the typed dependency (line 3 in Algorithm 1). For example, in Fig. 4, verb "Return" is extracted. The root word usually is the main verb of a sentence. We use the root word to help use locate the personal information by extracting the direct object of the root word using function *ExtractObj* (line 4 in Algorithm 1). In Fig. 4, the direct object of verb "Return" is the noun "list". It is worth noting that the direct object of the root word is only one word. If the object word is modified by other adjective words or nouns, we will also extract them and put them before the direct object in order to get a complete phrase. For instance, for the noun phrase "device ID", if "ID" is the direct object, we will extract "device" and put it before "ID".

Then, we use the personal information obtained from the method name and the class name as a reference to check if the direct object (i.e., obj) contains enough information. If there exists personal information in the method name, we extract it as the reference information (line 6-7). If the method name is a verb (i.e., it does not contain any personal information), we use the personal information extracted from the class name as the reference information (line 8-9). If neither the method name nor the class name contains personal information, the

Algorithm 1 Identifying the Personal Information Accessed by an API

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

```
1  desc_tree = StanfordParser_Tree(desc);
2  desc_dept = StanfordParser_Dept(desc);
3  root = ExtractRoot(desc_dept);
4  obj = ExtractObj(desc_dept, root);
5  nameInfo = null;
6  if Exist(name_method) then
7  |  nameInfo = name_method;
8  else
9  |  nameInfo = name_class;
10 end
11 if nameInfo != null then
12 |  simValue = Similarity(obj, nameInfo);
13 |  if simValue > threshold then
14 |  |  return obj;
15 |  else
16 |  |  ConatinName=obj.contain(nameInfo);
17 |  |  if ContainName == true then
18 |  |  |  return obj;
19 |  |  else
20 |  |  |  info=FindInfo(desc_tree, desc_dept);
21 |  |  |  obj = obj + info;
22 |  |  |  return obj;
23 |  |  end
24 |  end
25 else
26 |  return obj;
27 end
```

algorithm just returns the direct object (i.e., obj) as the final personal information (line 26).

After getting the reference information (i.e., $nameInfo$) from the method name and the class name, we first determine if additional postpositive attributives need to be added after the direct object. This is done by calculating the semantic similarity between direct object obj and reference information $nameInfo$ (line 12). If the semantic similarity value ($simValue$) calculated by ESA [21] is larger than the threshold (line 12-14), we think that the reference information can be found in the direct object, and then we return the direct object as the personal information accessed by this API. If the similarity value is lower than the threshold but the distinct words of the reference information appear in the direct object, we still think that the direct object contains the reference information (line 16-18). Otherwise, we regard that the direct object is not enough and some postpositive attributives should be added to make it more complete. In Fig. 4, the similarity value between the direct object "list" and the reference information "Running App Processes" is lower than threshold. At the same time, "Running App Processes" does not appear in the direct object. Therefore, we add postpositive attributives after "list".

Function $FindInfo$ locates the postpositive attributives of the direct object (line 20). More precisely, we check the phrase structure tree. If the subtree of the direct object contains phrases like "of …"/"from …"/"that …"/"for …", these postpositive attributes are extracted. We add the postpositive attributes (i.e., $info$) after the direct object to get the personal information (line 21-22). In Fig. 4, the postpositive attributive "of application processes" is added after "list". Finally, "list of application processes" is returned as the personal information accessed by the API *getRunningProcesses()*.

We use Explicit Semantic Analysis(ESA) [21] to compute the semantic relatedness between texts (line 12). Given two documents, 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 [22] because WordNet can only calculate the similarity between distinct words. Instead, ESA can compute the relatedness between arbitrarily long texts. In Fig. 4, since the semantic relatedness between obj "list" and "Running App Processes" calculated by ESA is lower than the threshold (0.5 in our system), obj cannot be regarded as the personal information, and thus more information should be added.

After mapping APIs to the personal information, we manually group APIs that request the same kinds of personal 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 III-E).

B. Static Analysis

Given an app, static analysis module first identifies invoked sensitive APIs and then determines the following information for each API:

- The personal information accessed by this app.
- Third-party libraries used in this app and the private information accessed by them.
- The condition(s) under which the personal information is accessed.
- Whether the personal information is retained or not.
- Whether the personal information is transferred or not.

Our static analysis is based on Vulhunter [12], which can construct an App Property Graph (APG) for each app and store it in the Neo4j graph database [23] for further processing. APG is a combination of AST (Abstract Syntax Tree), MCG (Method Call Graph), ICFG (Inter-procedural Control Flow Graph), and SDG (System Dependency Graph).

1) Static Analysis Module Overview: Algorithm 2 describes the basic procedure of our static analysis module. After enumerating invoked APIs, we first identify sensitive APIs/URIs, and then do reachability analysis on these sensitive APIs to remove infeasible code. After that, we map the selected APIs/URIs to personal information, and determine whether the private information is accessed by the host or third-party libraries. For reachable sensitive API calls, we extract the conditions of invoking them. Moreover, we conduct information

Algorithm 2 The Procedure of the Static Analysis Module

```
Input: stmt: statement in code
api = ExtractCalledAPI(stmt);
sensitive = SensitiveAnalysis(api);
reachable = ReachabilityAnalysis(api);
if sensitive == true ∧ reachable == true then
    Info = MapApiToInformation(api);
    User = UserIdentification(api);
    Conditions = ConditionExtraction(api);
    RetainOrNot = RetentionAnalysis(api);
    TransferredOrNot = TransferAnalysis(api)
    return (Info, User, Condition, RetainOrNot,
    TransferredOrNot);
end
return null;
```

retention analysis to find the personal information stored in file/log or sent out through network/SMS. We also perform information transfer analysis to determine if the information will be sent between the host app and third-party libraries or not. We detail these steps in the following paragraphs.

2) Sensitive API/URI Identification: The app can access sensitive information through the APIs/URIs described in Section III-A.1. In order to identify the used APIs/URIs, the static analysis module considers two kinds of statements: (1) statements that call sensitive APIs; (2) statements that access content providers (e.g., *ContentResolver.query()*). For the statements in the second case, we perform backward data flow analysis to find the sensitive URIs used as the parameters.

3) Reachability Analysis: If a statement calls sensitive APIs or uses sensitive URIs, we conduct reachability analysis to determine whether it is included in infeasible code [24], [25]. Since the infeasible code will not be executed, if the generated privacy policy describes such behaviors, it will raise false alerts. Section IV-B reports the experimental results of using reachability analysis to avoid false sentences.

To check if the sensitive API/URI is used in feasible code, we perform depth first traversal on the method call graph. If there is an execution path from entry points to the methods invoking sensitive APIs/URIs, the corresponding statement is reachable. Since Android apps have multiple entry points [26], such as life-cycle methods of each component and the callbacks of UI evnets, AutoPPG takes them into account.

4) Mapping APIs/URIs to Personal Information: The document analysis module creates a mapping between each API and the personal information. We use it to map the called sensitive API to corresponding personal information.

We map the selected URIs (i.e., the 8 URI strings and 160 URI fields) to the corresponding personal information manually through the required permissions. For example, since the URI "content://com.android.calendar" requires permission `READ_CALENDAR`, we map it to calendar. We do not use information extraction technique to automatically process these URIs' descriptions because most of them do not contain useful information. More precisely, after checking 160 URI fields, we find that 61 URI fields do not have descriptions in the official document. Although 46 of them have descriptions, they are useless. For example, the description of the field `<android.provider.Telephony$Mms$Inbox: android.net.Uri CONTENT_URI>` is "The content:// style URL for this table.". Only 53 URI fields have detailed descriptions, which could be processed automatically.

5) APIs/URIs User Identification: In order to find the embedded third-party libraries and determine whether they collect personal information, we record the classes having the statements that call sensitive APIs or use selected content providers. If a class name is the same as a third-party library's class name, we deem that the third-party library is the user of the personal information.

6) Condition Analysis: The execution conditions provide context information of invoking sensitive APIs. For each sensitive API/URI, AutoPPG identifies six kinds of conditions.

Five conditions are motived by AppContext [27] and DESCRIBEME [28], including: system events, device status, natural environment requirements, UI events, and hardware events. AppContext extracts four kinds of context factors while DESCRIBEME considers three kinds of conditions. Besides including these five conditions in AutoPPG, we also identify another kind of condition: device specific information, which includes the used language, screen size, OS version, etc.

▷ System events. Broadcast receivers can register for intents to be aware of the state changes of the system. For example, after booting, the system will broadcast the intent with *android.intent.action.BOOT_COMPLETED* action. If the device is on low battery, intent with *android.intent.action.BATTERY_LOW* action will be broadcasted. AutoPPG contains a list of 54 different kinds of intents. If sensitive APIs are called after the app receives certain intents, AutoPPG records the action of these intents.

▷ Device status. Android provides APIs to check the status of current device. For example, developers can call *PowerManager.isScreenOn()* to determine whether the device is in the interactive state or not. To collect all APIs related to device status, we first parse the official API document to get APIs fulfilling two requirements: the prefix of the method name is "*is*" and the type of return value is boolean. 640 APIs are found in this step. Since not all such APIs are relevant to system status, such as *URLUtil.isValidUrl(java.lang.String)*, we manually check them and keep 98 APIs.

AutoPPG checks all branch statements on the path from entry points to sensitive APIs and then conducts program slice based on data dependency relation. All APIs related to device status in these statements will be recorded.

▷ Natural environment requirements. Apps can check the current time or geolocation to perform different actions. Similar to APIs related to device status, AutoPPG also checks APIs relevant to time (e.g., *Date.getHour()*) and geolocation (e.g., *Location.getLatitude()*), and records them if they appear on the execution paths from entry points to sensitive APIs.

▷ Device specific information. Developers can check the language, screen size, and OS version. For example, *Locale.getDisplayLanguage()* returns the language suitable for display to the user. *Display.getSize()* obtains the display size in pixels. *Display.getHeight()* and *Display.getWidth()*

TABLE I
CALLBACK FUNCTIONS AND LISTENER REGISTER FUNCTIONS FOR VIEW CLASS

Callback function	Listener register function
onClick()	setOnClickListener()
onLongClick()	setOnLongClickListener()
onFocusChange()	setOnFocusChangeListener()
onKey()	setOnKeyListener()
onTouch()	setOnTouchListener()
onCreateContextMenu()	setOnCreateContextMenuListener()

```
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());
        }
    ...
}
```

Fig. 5. *com.abukai.expenses* obtains the location and saves it into a file.

return the height and width of the display, respectively. *android.os.Build.VERSION* holds various information about the OS and *SDK_INT* indicates the SDK version.

▷ UI events. It is a good practice for Apps to notify users when personal information will be collected, such as popping up a dialog. To identify such behaviors and write them into the privacy policy, AutoPPG extracts all UI callbacks that lead to the invocation of sensitive APIs. Apps can intercept event from touchscreen interface or keyboard-style input [29], [30]. For the former, developers can either statically define callback methods in the layout file (*e.g.*, through `android:onClick=""` attribute) or dynamically register event listener in code (*e.g.*, through *setOnClickListener()* in Tab. I). For the latter, developers can set the *OnKeyListener* for the *View* class and overwrite the callback method *onKey()* [31] or directly overwrite some methods of *View* class.

We parse the layout file to locate the callbacks defined in it. To find out the callbacks in the event listeners, we first locate all statements that register a listener, and then conduct data flow analysis to associate the view ID used in *findViewByID()* with the callback method in the corresponding listener. After finding the paths from the entry points to sensitive APIs, we record all UI callbacks on the paths. The texts appearing on the buttons are also extracted because they can help the privacy policy generator describe which button is pressed by the user.

▷ Hardware events. Users can press the BACK and HOME keys to affect the execution. For example, when the user presses the HOME key, the *onPause()* of current activity is called. When the user presses the BACK key, the *onResume()* of last activity will be executed. If the entry points of sensitive APIs are *onPause()* or *onResume()*, we will also record them.

7) Information Retention Analysis: Some personal 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 [16]. 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/URIs to the selected sink functions based on the data dependency relation.

For example, in app *com.abukai.expenses* (shown in Fig. 5), the class *com.abukai.expenses.Uti* calls *getLongitude()* and *getLatitude()*, and writes the results to the file through FileOutputStream's *write()* function. However, its privacy policy does not mention recording user's location information.

8) Information Transfer Analysis: The personal information obtained by the host app may be transferred to third-party libraries [32]. For example, the Ad library Admob provides *AdRequest.Builder().setLocation()* to pass location data to it [33]. As another example, tool libraries may get personal information (e.g., location) to facilitate identifying problems. We conduct static taint analysis to determine if the sensitive information will be transferred from the host app to a third-party library or vice versa by checking the data flow from sensitive APIs/URIs to the user of such data.

C. Verb Selection

The verb is an important component of a sentence. After finding an app's behaviors of collecting personal information, we select proper verbs for generating the sentences. First, when the app obtains the information through APIs, we use "access", "use", or "check" as the main verb for most such APIs. But if an app accesses external storage, we conduct data flow analysis to determine its real operation (i.e., read or write) and then automatically select the verbs. More precisely, apps can invoke APIs, like *Context.getExternalFilesDir()* and *Context.getExternalFilesDirs()*, to get file objects. If the app uses the file object to create a new directory (*e.g.*, *File.mkdir()*) and then writes data to file (*e.g.*, *FileOutputStream.write()*) or delete file (*e.g.*, *File.delete()*), we use "modify" as the main verb. Otherwise, we use "read" as the main verb.

Second, when the app obtains information by querying content providers, we determine the main verb according to the actual operations. More precisely, if the app queries the content provider by using *ContentResolver.query()* or *ContentResolver.update()*, we adopt "read" as the main verb. If the information is inserted into a content provider (*e.g.*, by *ContentResolver.insert()*), we employ "modify" as the main verb. If the information in content provider is deleted by *ContentResolver.delete()*, we use "delete" as the main verb.

D. Privacy Policy Generator

1) Sentence Generation: We generate sentences following the guidelines of plain English [14], which allows readers to understand the messages easily. Plain English has an average sentence length of 15-20 words and prefers active verbs. Hence, we define the structure of each generated sentence as:

$$[precondition]subject\ verb\ object$$
$$[purpose][postcondition]$$

This structure contains the following six parts:

Subject is the collector of personal information. If it is a third-party lib, we use the lib's name as the subject; otherwise, we use "We" as the subject of sentence.

Verb is determined according to the APIs used in apps (Section III-C).

Object is the collected personal information.

Precondition is an optional element. If the sensitive information is obtained after checking four kinds of conditions described in Section III-B.6 (i.e., system events, device status, natural environment requirements, and device specific information), we add a precondition (i.e., "If …") before the sentence. We employ the official descriptions of system events to construct the pre-condition. For instance, the description of the system event *android.action.TIMEZONE_CHANGED* is "The timezone has changed". If the sensitive APIs/URIs are used when receiving this event, we will use "If the timezone has changed" as the precondition. Since the official descriptions of the status APIs usually start with "Returns true if …" or "Returns true iff …", we extract the sub-sentences after these prefixes and use them as the preconditions. For example, the description of the status API *PowerManager.isPowerSaveMode()* is "Returns true if the device is currently in power save mode.". If the app calls *PowerManager.isPowerSaveMode()* in its branch statements, we use "If the device is currently in low power mode" as the precondition.

Purpose is another optional element that describes why the app accesses the personal information, because users are very concerned about how the personal information will be used [34]. Currently, AutoPPG can infer the purposes of three kinds of personal information, including camera, location, and contact. More precisely, if the camera is utilized to take picture (e.g., calling *Camera.takePicture()* in Fig.6), we let the purpose be "to take picture". If the camera is employed to record video (e.g., invoking *MediaRecorder.setVideoSource()*), the purpose becomes "to record video". For the location information, if it is transferred to advertisement libraries(e.g., through *AdRequest.setLocation()* of the Admob), the purpose is set to "for target advertising". Otherwise, if the location is obtained by social network related libraries (e.g., Facebook SDK), the purpose becomes "for location-based social networks". For the contact information, if it is used by SMS related APIs (e.g., *SmsManager.sendTextMessage()*), we let the purpose be "to send SMS". If the contact information is sent through email related intent (e.g., the action is `ACTION_SEND` and the intent includes extra data `EXTRA_EMAIL`), the purpose becomes "to send email". We will investigate how to infer more purposes in future work.

Postcondition is the last optional element. It is used to describe two kinds of conditions described in Section III-B.6 (i.e., UI events and hardware events). For example, as shown in Fig.6, the API (i.e., *Camera.open()*) is triggered after a button is pressed(i.e., *onClick()*), we add "when you press * button" after the sentence.

2) Additional Sentences: After generating sentence for the statement that calls sensitive API/URI, we add additional sentence to describe whether this information will be retained/transferred or not. If the information is retained in file or log, we add a sentence saying "This information will be written to file/log" after the generated sentence. Similarly, if the information is sent out through internet or SMS, we add a sentence saying "This information will be sent out through internet/SMS". If the information is obtained by the host app and transferred to a third-party library, we add a sentence indicating that this information will be transferred to the corresponding third-party library. Similarly, if the information is collected by a third-party library and transferred to the host app, we add a sentence saying that "This information will be transferred to the host app".

```
public void onClick(View v)  {
  boolean cameratest = (camera == null);
  try  {
    camera = Camera.open();
    camera.setPreviewDisplay(surfaceview.getHolder());
    camera.startPreview();
    camera.takePicture(shutterCallBack, rawCallBack, jpgCallBack);
  }
  catch(Exception e)  {
    Log.v(TAG, "Exception " + e.toString());
  }
}
```

Fig. 6. Code snippet of an app using camera.

3) Paragraph Generation: A privacy policy contains multiple parts. We also divide the generated sentences into four sections by referring the privacy policy template [35].

Personally Identifiable Information: We put all sentences about personally identifiable information in this section. According to [36] and [37], the personally identifiable information includes: account, IP address, MAC address, device ID, SIM card number, phone number, contact, call log, voice/microphone, camera/audio, SMS, calendar, location, country name, postal code, installed application list, visited URIs, and browser bookmarks.

Non-Personally Identifiable Information: This section contains sentences describing behaviors that are related to the non-personally identifiable information, including network type and browser type.

Cookies: If an app uses cookies to identify the user, we add a paragraph that describes such behaviors. More specially, if the app invokes cookies-related APIs, such as *CookieManager.getCookie(String)*, a sentence "*Cookies are files with small amount of data, which may include an anonymous unique identifier. Cookie will be used by:[User]*" will be added into the generated privacy policy. The *User* here is the app or the third-party libs that call cookie-related APIs.

Third-Party Lib and Information Disclosed to Them: If an app uses third-party libraries that access personal information, this section contains sentences relevant to such behaviors.

E. Post-Process

The generated document may contain duplicate sentences in a random order. To improve the readability, we perform two additional processing: removing duplicate sentences and changing the order of sentences.

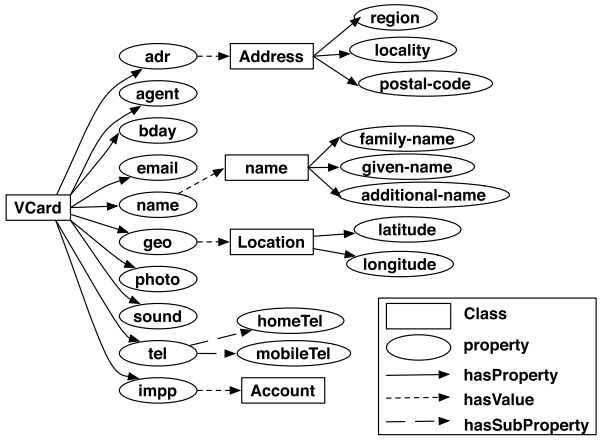

Fig. 7. Personal information ontology from *http://www.w3.org/TR/vcard-rdf/*

1) 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) 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) The information obtained by one API covers the information gained 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 duplicate sentences in the first case, we group APIs requesting the same kind of information together. When generating sentences, if multiple APIs in the same group are called by the same collector, the corresponding sentence will be generated only once. We also group those content provider's URI strings and URI fields to achieve the same purpose.

To remove duplicate sentences in the second case, we build a tree structure for personal information. Note that if the information on the parent node is collected, the information collected in its sub-tree is covered. We use a personal information ontology [38], [39] to build up such model. Ontology is a formal representation of a body of knowledges within a given domain [40]. The personal information ontology [38] covers all personal information, and organizes the personal information in a hierarchy structure (Fig. 7). We put APIs and URI (fields) into the corresponding classes and properties of this ontology. For example, *getLastKnownLocation()* is put into the class node "location", and *getLatitude()* is put into the property node"latitude". Then, we perform the depth first search on the ontology. If both a parent node's APIs and its child nodes' APIs are called, we remove the sentences resulted from the child nodes' APIs, and add their personal information into the the sentence resulted from the parent nodes' APIs.

2) Rearranging Sentences: After generating the sentences, we rearrange them according to the importance degree of the corresponding personal information. According to the "Upset Rate" for different risks studied in [41], we have a risk rank for different kinds of personal information (Table II). Based on this rank, we change the appearance order of sentences. For the information (*e.g.*, cookie) which does not appear in [41], we put them at the end of the privacy.

TABLE II
PERSONAL INFORMATION RISK RANK LIST

#	Personal Information	#	Personal 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		

Information Collection And Use
While using our application, this application will collect some personal information from your device.
(*) Personally Identifiable Information Collected by this Application.
We would access unique device ID. This information will be written to log.
We would use location (including, latitude and longitude).
If Wi-Fi is enabled, we would access mac address of wifi and IP address.
If the device is in an interactive state, we would check running tasks.
(*) Non-Personally Identifiable Information Collected by This Application.
If Wi-Fi is enabled, we would check network type (e.g.,GPRS, HSPA, LTE, UMTS).

Cookies
Cookies are files with small amount of data, which may include an anonymous unique identifier. Cookie will be used by this app.

Third Party Library and Information Disclosed to them
The following third party libraries are used by this application : Millennial, AdWhirl, Admob(Google), Flurry Analytics.
These third party libraries will also collect some information:
If network connectivity exists, Millennial would access unique device ID.
AdWhirl would use location (including, latitude and longitude).
Admob(Google) would access latitude and longitude.
Flurry Analytics would use location (including, latitude and longitude).

Fig. 8. Sample document generated by AutoPPG.

Fig. 8 shows a sample privacy policy generated by AutoPPG. Note that in this app we do not find any information that will be disclosed to third-party lib.

IV. EVALUATION

We conduct extensive experiments to answer the following research questions.

- **RQ1-Correctness:** How is the accuracy of AutoPPG's document analysis module and static analysis module?
- **RQ2-Readability:** How is the readability of the generated privacy policies?
- **RQ3-Adequacy:** Does the privacy policies generated by AutoPPG inform users all collected personal information?
- **RQ4-Developer Study:** How do the developers perceive the usability of AutoPPG?

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

A. 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.

TABLE III
THE APPS UNDER MANUAL CHECKING

Num	App	Num	App
1	air.com.goodgamestudios.empirefourkingdoms	11	com.jb.gokeyboard.plugin.emoji
2	com.gentaycom.nanu	12	com.rapidcyber.intelliim.android
3	com.king.petrescuesaga	13	com.rovio.angrybirdsgo
4	com.lego.juniors.quest	14	com.sharkparty.slots
5	com.microsoft.office.officehub	15	gpc.myweb.hinet.net.PopupWeb
6	jp.naver.SJLGBUBBLE	16	jp.konami.pesm
7	com.mobileiron	17	jp.naver.lineplay.android
8	air.com.sgn.cookiejam.gp	18	jp.naver.linetools
9	air.com.thomsonreuters.cnvne	19	org.asiministries.asievangelism
10	com.facebook.orca	20	si.modula.android.instantheartrate

It is worth noting that the comparison between generated privacy policies and existing privacy policies requires time-consuming manual verification, which cannot be done for a large number of apps. Therefore, for such experiments, we randomly select 20 apps that are listed in Table III.

B. Answer to RQ1: Correctness

We evaluate the correctness of AutoPPG from three aspects, including mapping APIs to their collected information, reachability analysis, and condition analysis.

1) Mapping APIs to Their Collected Information: To evaluate the accuracy of mapping APIs to their collected information, we randomly select another 150 functions from Susi [16] other than the 79 APIs mentioned in Section III-A.1. The document analysis module takes their signatures and descriptions as input, and we verify the output manually.

We found that 142 out of 150 APIs have correct results. The precision of our document analysis module is computed by the following formula:

$$precision = \frac{True\ Positive}{Total} = \frac{142}{150} = 94.7\%,$$

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.

8 out of 150 APIs are mapped incorrectly. The errors are caused by two reasons. One is due to the structure of the phrase structure tree generated by Stanford Parser. 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 identifies 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 the object "a string array", and our algorithm only searches postpositive attributes in the subtree of the object. The other reason is the threshold of semantic similarity. For example, the description of the API *Address.getAddressLine(int)* is: "*Returns a line of the address numbered by the given index …*". AutoPPG identifies the object "line" of the verb "Returns", but misses "of the address". It is because the semantic similarity between the object "line" and the information extracted from method name (i.e., "Address Line") is larger than the threshold and hence AutoPPG does not extract the additional information.

2) Reachability Analysis: We randomly select 16 apps that request permission `ACCESS_FINE_LOCATION` and call two location-related APIs (i.e., *getLatitude()* and *getLongitude()*), and manually verify the results from AutoPPG. In two apps, all statements that call these two APIs are reachable from entry points. 13 apps have both feasible and infeasible code. For example, the app *com.dooing.dooing* calls *getLatitude()* in two methods *<com.dooing.dooing.NewTask: void m()>* and *<com.dooing.dooing.cg: void a()>*. However, the latter method is never invoked by other methods. In one app (i.e., *com.smartsol.congresoandroid*, all statements that call these two APIs are infeasible. In total, the reachability analysis can successfully remove 89 false sentences due to infeasible code.

3) Condition Analysis: We compare AutoPPG with AppContext [27] by instructing them to process the apps shown in Table III. Here, we only examine the four kinds of conditions supported by AppContext. The result shows that AppContext finds out 84 APIs and their corresponding context information. AutoPPG identifies 83 of these 84 APIs. The missed one is *<android.hardware.Camera: android.hardware.Camera open()>* in the app *jp.konami.pesm*. We manually check it and find that this API is used in a third-party lib. However, the app will not execute the code because it does not request the permission `CAMERA` in its manifest file.

C. Answer to RQ2: Readability

We first evaluate how many duplicate sentences can be removed and then employ crowdsourcing to assess the readability of the generated privacy policies.

1) Removing Duplicate Sentences: We count the number of generated sentences in each app before and after removing the duplicates. As shown in Fig. 9, the curve with cross is on the left of the curve with square, meaning that duplicate sentences exist in many apps. More precisely, without deduplication, AutoPPG generates 5,782 sentences for 500 apps in total. After removeing duplicate sentences, only 3,991 sentences are left. In other words, 30.9% of sentences have been eliminated.

2) Questionnaire: For apps in Table III, we ask users to read the existing privacy policies and the ones generated by AutoPPG and answer two questions for each privacy policy.

Q1: "Do you think the above privacy policy is easy to read?". We provide four possible answers, including "Very difficult", "Difficult", "Easy", and "Very easy".

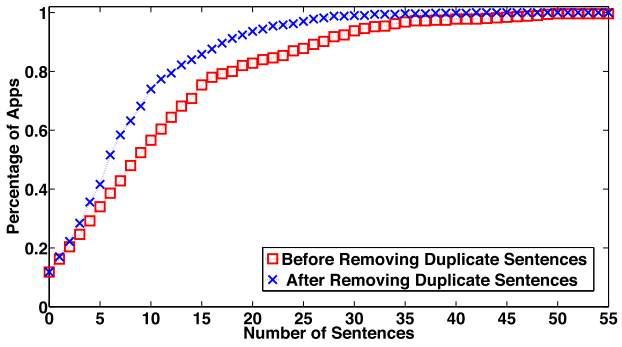

Fig. 9. Number of sentences before and after removing duplicate sentences

TABLE IV
AGE DISTRIBUTION OF WORKERS

Age	<20	20-30	30-40	40-50	50-60	>60
Number	0	6	11	7	4	2

TABLE V
DEGREE DISTRIBUTION OF WORKERS

Degree	High school	Some college	Associate	Bachelor	Graduate
Number	1	8	4	10	7

TABLE VI
THE MAP BETWEEN THE RESULT OF QUESTIONNAIRE AND SCORE

Answer	Score	Answer	Score
"Very Difficult"	1	"Difficult"	2
"Easy"	3	"Very Easy"	4

Q2: "If you select (Very) Difficult for the last question, could you please let us know the detailed reason?". We also prepare four possible reasons, including "Too long to read", "Unstructured privacy policy", "Privacy policy is unclear", and "Privacy policy is imprecise", and a comment box where users can provide other reasons.

All privacy policies and the questions are in one questionnaire. To avoid bias, we anonymize their names so that workers in *Amazon Turk* cannot identify the existing and generated ones. Moreover, we randomize the appearance order of these privacy policies before publishing them.

3) Background of Workers: After publishing them on Amazon Turk for 3 days, we received 30 responses. Note that each privacy policy in the questionnaire has been read by the same 30 workers. Table IV and Table V list the ages and obtained degrees of these workers, respectively. Most workers are 20-50 year old and hold at least high school degree. 19 of them are male and other 11 workers are female.

4) Result: After getting the answers to Q1, we map them to different scores as shown in Table VI, and get two data sets: one for the original privacy policies and one for the generated ones. Then, we apply two non-parametric hypothesis tests to the data sets. More precisely, to determine whether the two data sets are drawn from the same distribution, we use the Two-Sample Kolmogorov-Smirnov test (K-S) [42].

TABLE VII
THE HYPOTHESES OF TWO-SAMPLE KOLMOGOROV-SMIRNOV TEST

Null Hypotheses (H_0)	Two data sets are drawn from the same distribution.
Alternative Hypothesis (H_A)	Two data sets are **NOT** drawn from the same distribution.

TABLE VIII
THE HYPOTHESES OF MANN-WHITNEY U-TEST

Null Hypotheses (H_0)	The mean ranks of the two groups are equal.
Alternative Hypothesis (H_A)	The mean ranks of the two groups are **NOT** equal.

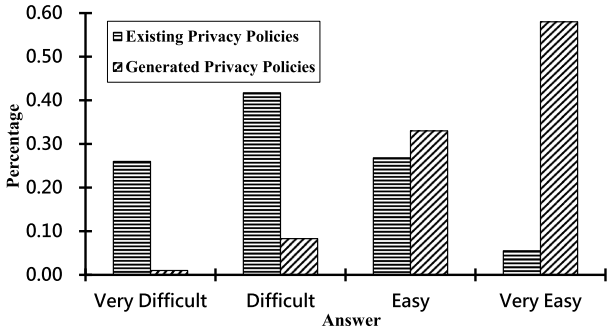

Fig. 10. Readability comparison between the original privacy policies and the generated privacy policies.

The hypotheses of K-S test are listed in Table VII. To test if the median values of the two data sets are the same, we adopt the Mann-Whitney U-test (MWU) [43]. The hypotheses of MWU test is listed in Table VIII.

Answer to Q1: Fig.10 shows the distribution of all responses to Q1. We can see that for the generated privacy policies around 90% readers choose "Easy" and "Very easy" whereas more than 65% readers select "Difficult" or "Very Difficult" for the original privacy policies. Table IX lists the results of hypothesis testing.

For K-S test, the p-value is the possibility that the two data sets are drawn from the same distribution. If the p-value is less than 0.05, the result of h = 1 indicates that we can *reject* the null hypothesis and accept the alternative hypothesis: the readability scores of the generated privacy policies and those of existing privacy policies have different distributions.

For MWU test, the p-value means the probability that two data sets have the same median. If the p-value is smaller than 0.05, the result of h = 1 denotes that we can *reject* the null hypothesis and accept the alternative hypothesis: the readability scores of the generated privacy policies and those of existing privacy policies have different medians.

The last column of Table IX lists the readability scores' average values. It shows that the generated ones have larger values than the existing ones in all except the 4th app. We will detail the reason when presenting the answers to Q2.

Answer to Q2: After analyzing the answers to Q2, we summarize the reasons why some existing privacy policies and generated ones are unreadable. For the original privacy

TABLE IX
THE NON-PARAMETRIC TEST RESULT AND EXCEPTION OF THE READABILITY SCORE OF EXISTING/GENERATED PRIVACY POLICIES

App Num	Kolmogorov-Smirnov Test	Mann-Whitney U-Test	Expectation Existing/Generated
1	p=0.001746, h=1	p=0.000053, h=1	2.47/3.27
2	p=0.000005, h=1	p=0.000000, h=1	1.97/3.30
3	p=0.000000, h=1	p=0.000000, h=1	1.90/3.70
4	p=0.342034, h=0	p=0.097875, h=0	3.47/3.27
5	p=0.000202, h=1	p=0.000063, h=1	2.57/3.43
6	p=0.000202, h=1	p=0.000008, h=1	2.23/3.30
7	p=0.000018, h=1	p=0.000000, h=1	2.00/3.30
8	p=0.000000, h=1	p=0.000000, h=1	1.50/3.60
9	p=0.001746, h=1	p=0.000011, h=1	2.33/3.23
10	p=0.000000, h=1	p=0.000000, h=1	1.50/3.80
11	p=0.000000, h=1	p=0.000000, h=1	2.00/3.90
12	p=0.000000, h=1	p=0.000000, h=1	2.73/3.73
13	p=0.000000, h=1	p=0.000000, h=1	1.73/3.47
14	p=0.000000, h=1	p=0.000000, h=1	1.47/3.83
15	p=0.000000, h=1	p=0.000000, h=1	2.57/3.80
16	p=0.000000, h=1	p=0.000000, h=1	1.83/3.67
17	p=0.000000, h=1	p=0.000000, h=1	1.27/3.37
18	p=0.000202, h=1	p=0.000000, h=1	2.30/4.43
19	p=0.000000, h=1	p=0.000000, h=1	2.60/3.73
20	p=0.000616, h=1	p=0.000018, h=1	1.93/2.77

policies, the major issue raised by the workers is that the original privacy policies are too long to read. In particular, they contain much unimportant information but less information about the collected data. Besides, many workers think that the original privacy policies are unstructured. For instance, after reading the 2nd app's original privacy policy, one worker wrote "The information regarding sharing information with third parties is in the middle, easy to miss". There are also many workers regarding that the original privacy policies are unclear. For example, after reading the 1st app's original privacy policy, one worker left "In one place it says it is up to the user to provide personal information, and in another it says it must be provided to participate in certain things". Furthermore, some workers feel that existing privacy policies are imprecise.

In contrast, although some workers also think that the generated privacy policies have similar issues, the number of such comments is much less than that for the original privacy policies as shown in Fig.10. More precisely, some workers feel that the generated privacy policies are unclear because they cannot understand the technical words in it. For example, after reading the generated privacy policy for the 3rd app, one worker said "*There is some technological jargon that makes it a bit hard to read*". There are some workers thinking that several generated privacy policies are unstructured. For instance, AutoPPG identifies five third-party libraries in the 20th app, and describes their behaviors in the same paragraph. Therefore, workers think that it is unstructured. Besides, several workers regard that a few generated privacy policies are too long. As an example, AutoPPG finds that the 4th app has two third-party libraries that collects multiple kinds of information, and depicts their behaviors in the generated privacy policy. In contrast, the original privacy policy does not have such descriptions. Finally, a few workers feel that some generated privacy policies are imprecise. For example, the generated privacy policy for the 10th app says that device ID will be accessed without giving user additional information.

As how to construct effective privacy notes and display them is still an open problem [44], we will investigate it to enhance the privacy policy generated by AutoPPG in future work.

D. Answer to RQ3: Adequacy

We compare the coverage of the generated privacy policies on the collected personal information and that of the existing privacy policies for the apps in Table III. In particular, we focus on 10 kinds of personal information, including device ID, location, account, camera, audio, installed app list, cookies, phone number, call log, calendar, and count the number of third-party libraries mentioned in the privacy policies.

Table X lists the comparison result that have been manually verified. It shows that the generated privacy policies contain 45 kinds of personal information. However, existing privacy policies just list 29 kinds of personal information in total. In other words, AutoPPG identifies more personal information collected by apps than the existing privacy policies. Moreover, the generated privacy policies list 20 third-party libraries whereas the existing privacy policies only mention 4. For example, the 20th app's privacy policy only informs users that device ID will be used whereas the privacy policy generated by AutoPPG indicates that both device ID and camera will be used. Moreover, the 20th app's existing privacy policy does not mention third-party libraries whereas AutoPPG reveals 5 third-party libraries used by this app.

Based on Table X, we found two kinds of issues in existing privacy policies. First, some privacy policies are incomplete because they do not include all personal information accessed by the app. For example, 4 apps (the 1st, 6th, 10th, 17th app in Table X) use the application list in code but none of them declares it in the privacy policies. Second, some privacy policies contain certain personal information that is *not* obtained by apps. For example, the 14th app's privacy policy contains the sentence "we collect information regarding your device type, operating system and version, …, the device's geo-location". However, the app does not request location related permission in the manifest, meaning that it will not access location information. AutoPPG can avoid these issues because it uses code level information to generate privacy policy.

E. Answer to RQ4: Developer Study

To understand how the developers perceive the usability of AutoPPG, we collect the developers' email addresses from Google Play and ask them to finish a survey, which contains a privacy policy generated by AutoPPG and the questions (i.e., **Q1**-**Q4**). We sent emails to 3,700 developers and only received 45 responses in 3 days as developers may not check them frequently. For example, some emails addresses may be used for Q&A because they reply with FAQ automatically. The results show that most developers, who reply our emails, would like to use AutoPPG to facilitate them.

Q1: "*If our tool is available to generate the privacy policy template for your app, would you like to use the tool?*".

TABLE X
COMPARING THE COVERAGE OF GENERATED PRIVACY POLICIES AND THAT OF EXISTING PRIVACY POLICIES. "N": NEW PRIVACY POLICIES, "O": OLD PRIVACY POLICIES

App Num	Device ID		Location		Account		Camera		Audio		App List		Cookies		Phone Num		Call log		Calendar		Lib Num	
	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O	N	O
1			√								√										2	1
2	√			√											√		√				2	
3	√												√	√							1	
4		√	√	√			√														2	1
5					√	√																
6	√										√		√									
7	√		√											√	√		√					
8	√		√										√	√							2	
9		√	√				√						√	√							2	
10	√	√	√	√							√					√						
11																						
12							√												√			
13		√	√	√			√					√	√	√							2	1
14	√	√		√										√								
15	√	√																				
16			√										√	√								
17	√	√		√			√				√		√	√							1	1
18				√			√						√									
19		√		√					√				√								1	
20	√	√					√							√							5	
Total	10	9	8	8	1	1	7	0	1	0	4	1	9	9	2	1	2	0	1	0	20	4

26.7% developers (12/45) selected "Absolutely YES" and 53.3% developers (24/45) selected "YES". Only 20.0% developers (9/45) selected "NO". 3 developers wrote down the reasons, including, "too busy", "It doesn't interest me", and "nothing is saved besides the data entered by the user".

Q2: "*When you integrate third-party libraries in your app, do you know which privacy information will be accessed by them?*". We want to see if developers know the third-party libraries' behaviors related to privacy information. 42.2% developers (19/45) select "NO" and 2.2% developers (1/45) select "Absolutely NO". AutoPPG can help these developers profile such behaviors in third-party libraries.

Q3: "*Since third-party libraries seldom provide source code, would you like to use our tool to identify the third-party libraries' behaviors related to users' privacy information?*". 84.4% developers select "Absolutely YES" (9/45) and "YES" (29/45). 15.6% (7/45) developers select "NO" and 3 developers left the reasons: "I mostly use open source libs", "I use open source library", and "breaking someone else's trust and code".

Q4: "*If the development of your apps were outsourced, would you like to use our tool to generate the privacy policy template for your apps?*". This question is to determine if AutoPPG would be used when the development of apps is outsourced (i.e., the owner does not understand the source code). 84.4% developers select "Absolutely YES" (9/45) and "YES" (29/45). Only 15.6% developers (7/45) refused to use AutoPPG. One gave the reason: "If I was outsorsing the development, I would outsource the privacy policy preparation along with it, to the same entity". Unfortunately, even app developers may not be able to write correct privacy policies.

V. DISCUSSION

This section discusses the limitations of AutoPPG and our future work. First, we use the mapping between APIs/URIs to permissions provided by PScout [17]. Backes et al., recently found that PScout may lead to false mappings [45]. We will use the new mapping in [45] once it is available. Moreover, we will include more libraries' class names in the list to increase the accuracy of identifying third-party libraries.

Second, since the callback method is implemented by the app but invoked by the Android framework, the control flow is implicitly transferred to the callback method via the framework [46]. As AutoPPG only analyzes apps without examining the framework, it may miss some control and data flows. Currently, AutoPPG only handles the UI callbacks listed in the official document when conducting the condition analysis. In future work, we will leverage the methods proposed in [46] to obtain the complete control and data flows and then handle more callback methods.

Third, when checking the parameters to the content providers, AutoPPG currently cannot handle complex string operations such as "append", "split" [47]. Hence, AutoPPG may miss the use of some sensitive URIs. We will employ string analysis [48] to approach this problem in future work. Moreover, since AutoPPG only employs static analysis to identify apps' behaviors related to personal information, it may lead to false positives. We will incorporate dynamic checking to remove them in further work.

VI. RELATED WORK

We introduce closely related works on privacy policy, privacy policy generator, and software artifacts generation.

A. Privacy Policy

Since privacy policies are written in natural language, some studies investigated how to parse privacy policies and mine information from them. Breaux and Anton [49] present a methodology for directly extracting access rights and obligations from the Health Insurance Portability and Accountability Act (HIPAA) Privacy Rule. To automate the process, semantic patterns are employed to handle the access control policy [50], [51]. Recent studies [52], [53] compared the behaviors described in privacy policy with the actual behaviors in code to detect problematic privacy policies.

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 [54]. Zimmeck et al. combine crowdsourcing and machine learning techniques to check essential policy terms in privacy policies [55]. Liu et al. adopt HMM to perform the automatic segment alignment for policies [56], [57].

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

B. Privacy Policy Generator

Although Privacy Informer [8] 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 [58] 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 [2]. It detects privacy-related behaviors in apps and explains them to end users [2]. 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 personal 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.

This paper is an extended version of our previous workshop paper [59]. Comparing with [59], we extend the contents from the following aspects. First, we enhance the static analysis module by adding the reachability analysis, the information transfer analysis, and the enhanced condition analysis with six kinds of conditions. Note that the previous study [59] does not remove infeasible code and only analyzes one kind of condition. The enhancements can remove the false sentences and provide more detailed information to users.

Second, we propose a new template for generating privacy policy, which is much more expressive than the old one in [59]. More precisely, it includes six kinds of conditions found in code and two kinds of additional sentences for describing the information to be retained and/or transferred.

Third, we propose a new method to select verbs for personal information. Note that the previous study [59] selected the verb for personal information by analyzing existing documents. However, the selected verb may not reflect the real behaviors of the apps. In this paper, for most APIs, we manually define verbs for the corresponding personal information. Moreover, if an app accesses external storage through APIs, we conduct data flow analysis to determine its real operation (i.e., read or write) and then automatically select the verbs. Similarly, if the app obtains information by querying content provider with URIs, AutoPPG determines the verb according to the operation (e.g., query, insert, delete).

Fourth, we conduct more evaluations on the enhanced AutoPPG. More precisely, we compare the sensitive APIs found by AutoPPG and AppContext, and evaluate the reachability analysis. We conduct new experiments to evaluate the readability of the generated privacy policies, where the questionnaires are carefully designed to avoid bias and obtain more information about the workers. Besides, we use hypothesis testing to analyze the results and find that the privacy policies generated by AutoPPG have a better readability. Moreover, we also study how the developers perceive the usability of AutoPPG and find that most developers, who reply our emails, would like to use AutoPPG to facilitate them.

C. 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 designed model checker can produce complete test sequences that provide a pre-defined coverage of any software artifact [60]. 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 [61]. 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 [62]. 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 [63].

Zhang et al. propose the system DESCRIBEME to generate security-centric descriptions for Android apps [28]. The differences between DESCRIBEME and AutoPPG include: (1) They have different purposes. More precisely, DESCRIBEME aims at generating descriptions while AutoPPG intends to help developer create privacy policy. (2) DESCRIBEME only considers sensitive APIs while AutoPPG analyzes both sensitive APIs and URIs. (3) DESCRIBEME can only extract three kinds of conditions for sensitive APIs while AutoPPG can extract five kinds. (4) DESCRIBEME identifies high-level patterns from behavior graphs to reduce the size of generated description. Instead, AutoPPG first generates sentences for all behaviors relevant to personal information and then removes duplicate sentences and employs the ontology graph to combine remaining sentences. Moreover, AutoPPG divides the generated sentences into multiple parts and rearrange the order of generated sentences.

VII. 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 personal 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' personal 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, the readability and the adequacy of the generated privacy policies. Moreover, most developers who answer our survey would like to use AutoPPG to facilitate them. In future work, besides further improving the system, we will examine more apps and involve more persons to evaluate AutoPPG.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their quality reviews and suggestions.

REFERENCES

[1] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android permission model and enforcement with user-defined runtime constraints," in *Proc. ASIACCS*, 2010, pp. 328–332.

[2] 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, pp. 221–232.

[3] *Trusted legal Agreements*, accessed on 2016. [Online]. Available: https://termsfeed.com

[4] "The need for privacy policies in mobile apps—An overview," iubenda s.r.l," FTC Staff Report, Tech. Rep., 2013.

[5] (2013). *The Need for Privacy Policies in Mobile Apps—An Overview*. [Online]. Available: http://goo.gl/7AB2aB

[6] "Path social networking app settles ftc charges it deceived consumers and improperly collected personal information from users' mobile address books," Federal Trade Commission, Tech. Rep. Case3:13-cv-00448-RS, 2013. [Online]. Available: https://goo.gl/Z31BAU

[7] C. Meyer, E. Broeker, A. Pierce, and J. Gatto. (2015). *FTC Issues New Guidance for Mobile App Developers that Collect Location Data*. [Online]. Available: http://goo.gl/weSNRB

[8] D. Y. Miao, "PrivacyInformer: An automated privacy description generator for the mit app inventor," M.S. thesis, Massachusetts Inst. Technol., Cambridge, MA, USA, 2014.

[9] *MIT App Inventor*, accessed on 2015. [Online]. Available: http://appinventor.mit.edu

[10] (2015). *Google Java Style*. https://goo.gl/1RtxN1

[11] C. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge U.K.: Cambridge Univ. Press, 2008.

[12] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: Toward discovering vulnerabilities in Android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44–53, Jan. 2015.

[13] C. Manning and H. Schutze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[14] M. Cutts, *Oxford Guide to Plain English*, 4th ed. Oxford, U.K.: Oxford Univ. Press, 2013.

[15] D. Cer, M. Marneffe, D. Jurafsky, and C. Manning, "Parsing to stanford dependencies: Trade-offs between speed and accuracy," in *Proc. LREC*, 2010, pp. 1–5.

[16] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *Proc. NDSS*, 2014, pp. 1–15.

[17] K. Au, Y. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the Android permission specification," in *Proc. CCS*, 2012, pp. 217–228.

[18] *Natural Language Toolkit*, accessed on 2015. [Online]. Available: http://www.nltk.org/

[19] J. Cowie and W. Lehnert, "Information extraction," *Commun. ACM*, vol. 39, no. 1, pp. 80–91, 1996.

[20] Y. Xu, M.-Y. Kim, K. Quinn, R. Goebel, and D. Barbosa, "Open information extraction with tree kernels," in *Proc. HLT-NAACL*, 2013, pp. 868–877.

[21] E. Gabrilovich and S. Markovitch, "Computing semantic relatedness using wikipedia-based explicit semantic analysis," in *Proc. IJCAI*, 2007, pp. 1–16.

[22] *WordNet: A lexical Database for English*, accessed on 2015. [Online]. Available: http://wordnet.princeton.edu/

[23] *Neo4j: The World's Leading Graph Database*, accessed on 2015. [Online]. Available: http://neo4j.com/

[24] S. Arlt and M. Schäf, "Joogie: Infeasible code detection for java," in *Proc. CAV*, 2012, pp. 767–773.

[25] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of Android applications' permissions," in *Proc. SERE*, 2012, pp. 45–46.

[26] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. PLDI*, 2014, pp. 259–269.

[27] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. ICSE*, May 2015, pp. 303–313.

[28] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for Android apps," in *Proc. CCS*, 2015, pp. 518–529.

[29] J. Ostrander, *Android UI Fundamantals: Develop Design*. Berkeley, CA, USA: Peachpit, 2012.

[30] I. Darwin, *Android Cookbook*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2012.

[31] *Android Developers: View.OnKeyListener*, accessed on 2015. [Online]. Available: http://goo.gl/hdaVg5

[32] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in Android ad libraries," in *Proc. MoST*, 2012, pp. 1–10.

[33] *AdMob by Google: Targeting*, accessed on 2015. [Online]. Available: https://goo.gl/c4uww1

[34] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang, "Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing," in *Proc. UbiComp*, 2012, pp. 501–510.

[35] (2015). *Generic Privacy Policy Template*. [Online]. Available: https://media.termsfeed.com/pdf/privacy-policy-template.pdf

[36] (2015). *Guide to Protecting the Confidentiality of Personally Identifiable Information (PII)*. [Online]. Available: https://goo.gl/gEzSoI, .

[37] R. V. Connelly. *What is Personally-Identifiable Information (PII)*. [Online]. Available: http://goo.gl/lDpPsZ

[38] H. Halpin, B. Suda, and N. Walsh. (2015). *An Ontology for vCard*. [Online]. Available: http://www.w3.org/2006/vcard/

[39] R. Iannella and S. Identity. (2015). *vCard Ontology—For Describing People and Organizations*. [Online]. Available: http://www.w3.org/TR/vcard-rdf/

[40] (2015). *Ontology Structure*. [Online]. Available: http://goo.gl/sBGgzZ

[41] 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. SPSM*, 2012, pp. 33–44.

[42] *Two-Sample Kolmogorov-Smirnov Test*. [Online]. Available: http://goo.gl/j9z8CJ

[43] *Wilcoxon Rank Sum Test*. [Online]. Available: http://goo.gl/t6ihv2

[44] F. Schaub, R. Balebako, A. Durity, and L. Cranor, "A design space for effective privacy notices," in *Proc. ACM SOUPS*, 2015, pp. 1–17.

[45] M. Backes, S. Bugiel, E. Derr, S. Weisgerber, P. McDaniel, and D. Octeau, "On demystifying the Android application framework: Revisiting Android permission specification analysis," in *Proc. USENIX SEC*, 2016, pp.1101–1118.

[46] Y. Cao *et al.*, "EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework," in *Proc. NDSS*, 2015, pp. 1–15.

[47] F. Shen *et al.*, "Information flows as a permission mechanism," in *Proc. ASE*, 2014, pp. 515–526.

[48] J. D. Vecchio, F. Shen, K. M. Yee, B. Wang, S. Y. Ko, and L. Ziarek, "String analysis of Android applications (N)," in *Proc. ASE*, 2015, pp. 680–685.

[49] T. Breaux and A. Antón, "Analyzing regulatory rules for privacy and security requirements," *IEEE Trans. Softw. Eng.*, vol. 34, no. 1, pp. 5–20, Jan./Feb. 2008.

[50] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural language software documents," in *Proc. FSE*, 2012, Art. no. 12.

[51] J. Slankas, X. Xiao, L. Williams, and T. Xie, "Relation extraction for inferring access control rules from natural language artifacts," in *Proc. ACSAC*, 2014, pp. 366–375.

[52] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of Android apps?," in *Proc. IFIP/IEEE DSN*, Jun. 2016, pp. 538–549.

[53] R. Slavin *et al.*, "Toward a framework for detecting privacy policy violations in Android application code," in *Proc. ICSE*, Jun. 2016, pp. 538–549.
[54] E. Costante, Y. Sun, M. Petković, and J. Hartog, "A machine learning solution to assess privacy policy completeness," in *Proc. WPES*, Oct. 2012, pp. 91–96.
[55] S. Zimmeck and S. M. Bellovin, "Privee: An architecture for automatically analyzing Web privacy policies," in *Proc. USENIX Secur.*, 2014, pp. 1–16.
[56] F. Liu, R. Ramanath, N. Sadeh, and N. Smith, "A step towards usable privacy policy: Automatic alignment of privacy statements," in *Proc. COLING*, 2014, pp. 884–894.
[57] R. Ramanath, F. Liu, N. Sadeh, and N. Smith, "Unsupervised alignment of privacy policies using hidden Markov models," in *Proc. ACL*, 2014, pp. 605–610.
[58] M. Rowan and J. Dehlinger, "Encouraging privacy by design concepts with privacy policy auto-generation in eclipse," in *Proc. ETX*, 2014, pp. 9–14.
[59] L. Yu, T. Zhang, X. Luo, and L. Xue, "AutoPPG: Towards automatic generation of privacy policy for Android applications," in *Proc. ACM SPSM*, 2015, pp. 39–50.
[60] S. Rayadurgam and M. P. E. Heimdahl, "Coverage based test-case generation using model checkers," in *Proc. ECBS*, Apr. 2001, pp. 83–91.
[61] A. Z. Javed, P. A. Strooper, and G. N. Watson, "Automated generation of test cases using model-driven architecture," in *Proc. AST*, May 2007, p. 3.
[62] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proc. ESEC/FSE*, 2011, pp. 212–222.
[63] M. J. Rutherford and A. L. Wolf, "A case for test-code generation in model-driven systems," in *Proc. GPCE*, 2003, pp. 377–396.

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

Tao Zhang received the Ph.D. degree in computer science from the University of Seoul. He is currently a Research Associate with the Department of Computing, The Hong Kong Polytechnic University. He is also an Associate Professor with the College of Computer Science and Technology, Harbin Engineering University. He was an Assistant Professor with the School of Computer Science and Technology, NUPT. His research interests include mining software repositories and security for Android Apps.

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

Lei Xue received the B.S. and M.S. degrees in major of information security from the Nanjing University of Posts and Communications. He is currently pursuing the Ph.D. degree with the Department of Computing, The Hong Kong Polytechnic University. His research interest includes network security, network measurement, and mobile security.

Henry Chang was the IT Advisor to the Hong Kong Privacy Commissioner and the Founding Chair with the Technology Working Group, Asia Pacific Privacy Authorities. He is currently an Adjunct Associate Professor with the Law Department, The University of Hong Kong. He is also a Hong Kong appointed Technical Expert to the ISO Identity Management and Privacy Technologies Group.