GUILeak: Identifying Privacy Practices on GUI-Based Data

Xiaoyin Wang[1], Xue Qin[1], Mitra Bokaei Hosseini[1],
Rocky Slavin[1], Travis D. Breaux[2], and Jianwei Niu[1]
[1]University of Texas at San Antonio, San Antonio, TX, USA
[2]Carnegie Mellon University, Pittsburgh, PA, USA
{xiaoyin.wang, xue.qin, mitra.bokaeihosseini, rocky.slavin, jianwei.niu}@utsa.edu,
breaux@cs.cmu.edu

***Abstract*—As the most popular mobile platform, Android devices have millions of users around the world. As these devices are used everyday and collects various data from users, effective privacy protection has been a well known challenge in the Android world. Existing privacy-protection approaches focus on information accessed from Android API methods, such as location and device ID, while existing security-enhancement approaches are not fine-grained enough to map user input data to concepts in privacy policies. In this paper, we proposed a novel approach that automatically detects privacy leakage on user input data for a given Android app, and determines whether such leakage may violate privacy policies coming with the Android app. For evaluation, we applied our approach to 80 popular apps from two important app categories: finance and health. The results show that our approach is able to detect 20 strong violations, and 10 weak violations from the studied apps.**

I. Introduction

The Android mobile platform supports millions of users and their mobile devices across more than 190 countries around the world. In February 2016, the Google Play Store, which provides users access to new applications (apps), surpassed two million apps available for installation [1]. Increasingly, Android apps use personal information to provide services to users. According to a recent 2015 study, 500 million people were using personal health apps, which can collect information on body measurements, diet, exercise, and medical treatment, among others. Similarly, Mint, one of the most popular personal finance apps, tracks more than 80 billion dollars in credit and debit transactions, and almost a trillion dollars in loans and assets. What users of these health and finance apps may not understand, however, is how their personal information is collected, used and shared by these apps.

With increased access to personal information and the scale of mobile app deployment, the need for tools to help developers to protect user privacy is increasingly important. Google encourages app developers to provide users with privacy policies that describe how personal information is collected from users [24]. These policies are written in natural language and aim to describe the ways that an app collects, uses, and shares user data. Such policies are also meant to fulfill legal requirements to protect customer privacy. Prior work by Slavin et al. [24] traced privacy policy statements about collection to Android platform API calls using an ontology to link policy semantics with static program analysis. These API calls concern personal data that is automatically collected, such as user location, device identifiers, and sensor data. However, this work was limited because it did not address personal data that users provide directly through an app's user interface. In health and finance apps, this type of collection is especially sensitive, because only users can provide this data. User-provided information presents two new technical challenges that we address in this paper:

TC1: Vague and Unbounded User-Provided Information Types. The types of information collected through the Android API are limited by hardware and platform capability, thus a mapping from API methods to canonical policy terms is constrained and well-defined. In contrast, developers can design novel user interfaces that ask users to provide potentially any kind of information, which includes unstructured and semi-structured personal information in different formats.

TC2: Varying User Interface Implementation Techniques. Unlike API method calls that can be detected by scanning the app byte code, user interfaces can be implemented by static declarations in resource files, or programmatically in the code. Techniques, such as SUPOR [13] and UIPicker [17], can identify input views receiving sensitive user input, but they do not map these views to relevant policy terms, nor do they identify programmatically-generated input views.

In this paper, we present a novel technique to detect privacy-policy violations on user-provided information for Android apps. The approach maps each user interface (GUI) input view to terminology in privacy policies, and then performs static information flow analysis to detect illegal information flows in the app's code that violate relevant policy statements. To address **TC1**, For each input view in a new app, we use various phrase similarity measurements to map the graphics user interface (GUI) labels together with its context to ontology phrases. The ontology is further matches to the privacy policy text. To address **TC2**, we developed a GUI string analysis technique to estimate the structure of programmatically-generated UIs and collect all UI labels in the context of a given input view. Our analysis is based on GATOR [22], an existing GUI analysis framework.

To validate our approach, we focus on two app categories (domains) in the Google Play Store[1]: Health and Finance. These two categories are important, because they access sensitive personal information. Furthermore, these two domains are potentially regulated by the Gramm-Leach-Bliley Act (GLBA) [8], Right to Financial Privacy Act (RFPA) [11], Health Insurance Portability and Accountability Act (HIPAA) [21], and Payment Card Industry Data Security Standard (PCI DSS) [9]. In our experiment, we collected 100 of the most popular apps and their privacy policies (50 apps for each category), and we used 20% of the apps (10 apps from each category) as a training set. Using the privacy policies from the training apps, we constructed an ontology for each of the two domains. Then, we applied our approach to the remaining 80 apps, and detected 30 violations, which we manually confirmed by recording the runtime network requests with Xposed framework[2].

In this paper, we present four contributions as follows.

- We developed a novel approach including GUI analysis and phrase mapping techniques to detect inconsistencies between the information collection statements in an app's privacy policy and the actual collection behavior of user input data in the app's code.
- Using existing crowd sourcing tools, we developed privacy ontologies for two privacy-sensitive domains: health and finance, which contain 332 and 209 ontology terms, respectively.
- We carried out an experiment on 80 of the most popular apps in health and finance domains and detected 20 strong violations and 10 weak violations.

This paper is organized as follows: in Section III, we review the background upon which we based our approach and introduce relevant concepts; in Section IV, we describe the approach we used for our framework; Section V describes the evaluation of our approach followed by discussion of the results and approach in Section VI; Section VII includes related work; and we conclude and discuss future work in Section VIII.

II. Motivating Example

In this section, we give a real example showing how GUI views, especially user input views can be implemented in different ways, and the difficulties of understand input views.

Dynamic Layouts. A *layout* in the Android framework defines the visual structure of a UI including locations for views, buttons, windows, and widgets. Layouts are defined in two ways: either they are constructed using XML to define the placements of elements, or they are created programmatically at runtime. The second way is necessary when layouts need to be dynamically changed based on runtime states. These *dynamic layouts* eliminate the need to pre-draw multiple UIs. These two ways can be combined flexibly. For example, a label or view can be programmatically added to a static defined layout (typically after inflation).

[1]http://play.google.com
[2]http://repo.xposed.com

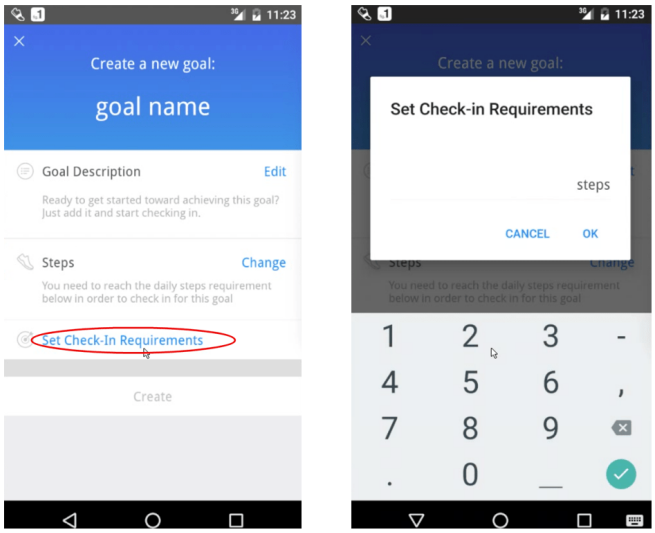


Fig. 1: User Interface Screenshots from Pacer

```
1       const v2, 0x7f07011a
2
3       .line 62
4       invoke-virtual {v1, v2}, Lcom/afollestad/materialdialogs/
5       MaterialDialog$Builder;->title(I)Lcom/afollestad/
6       materialdialogs/MaterialDialog$Builder;
7
8       move-result-object v1
```
Fig. 2: Virtual Invoke Example in GoalSetCheckingInReqDialog.smali

The example in Figure 1 shows two UI screenshots for Pacer, a popular fitness app, depicting the UI when user creates a new exercise goal. Besides editing goal descriptions and changing goal types such as steps and diet, the user also needs to set the check-in requirement by clicking the button in the red oval on the left screenshot. The right screenshot shows the pop-up window that appears after user clicking this button. Here, users will be asked to type in the desired number of check-in steps.

The right screenshot utilizes a combination of both static

```
1   <LinearLayout android:gravity="center_horizontal" ...>
2       <cc.pacer.androidapp.ui.common.fonts.TypefaceTextView
3           ... android:id="@id/title" ... />
4       <cc.pacer.androidapp.ui.common.fonts.TypefacedEditText
5           ... android:id="@id/et_content" ... />
6       <LinearLayout ...>
7           <Button ...
8               android:id="@id/btnLeft" ...
9               android:text="@string/btn_cancel" .../>
10          <Button ...
11              android:id="@id/btnRight" ...
12              android:text="@string/yes" ... />
13      </LinearLayout>
14  </LinearLayout>
```
Fig. 3: Partial Code from common_input_dialog.xml

```
1   <LinearLayout android:orientation="vertical" ...>
2       ...
3       <LinearLayout ... android:background="@drawable/goal_create_top_background"
        .. > ...
4        <cc.pacer.androidapp.ui.common.fonts.TypefaceTextView ...
5           android:id="@id/tv_goal_create_title" ... />
6       </LinearLayout>
7       ...
8       <RelativeLayout android:id="@id/rl_goal_create_details_set_requirement" ...>
9       ...
10      <cc.pacer.androidapp.ui.common.fonts.TypefaceTextView ...
11          android:text="@string/goal_set_requirements" ... />
12      ...
13      </RelativeLayout>
14      ...
15  </LinearLayout>
```
Fig. 4: Partial Code from goal_create_details_fragment.xml

and dynamic layouts. Figure 3 shows the static definition of `common_ input_dialog.xml`, which corresponds to the right screenshot. From the figure, we can see that, it is actually a *layout template* which defines the basic layout structure and the font / style information. All the labels (e.g., Dialog title is undefined) and ids (e.g., "et_content" as the id of the input box) are vaguely defined. Thus this layout template can be used in multiple places in the project for user input, and the labels (e.g., Set Check-In Requirements for title) will be transferred from the parent activity (e.g., the activity in the left screenshot) when the dialog is opened.

In particular, Figure 2 shows the smali code (decoded Android bytecode) in `InputDialogFragment.smali` which dynamically adds the label for "Set Check-In Requirement". The string is fetched in line 1 as `v2` with the id `0x7f07011a`. Here, the id references the appropriate string in `string.xml` based on the context. In line 4, `v2` is passed as a title resulting in "Set Check-In Requirement" being dynamically defined as the title.

GUI Context. Just like the contexts in natural language paragraphs, input views can only be well understood with neighboring / ancestor views. In the right screenshot, without seeing the title, it is not possible to understand what is supposed to be input. Furthermore, the left screenshot that leads to the right dialog also provides context information for the dialog. This invoking view can be found in the resource layout file `goal_create_details_fragment.xml`, as shown in Figure 4, whose id is .

To sum up, GUI context is essential in understanding user input views and mapping the views to privacy-policy phrases, but the dynamic implementation of Android GUI makes identification of GUI context more difficult. In our paper, we propose input context analysis to handle dynamic implementation and hierarchical mapping to map input views to privacy policy phrases based on collected GUI context.

III. Background

In this section, we introduce some background knowledge about the existing techniques upon which our approach is based.

A. GATOR

GATOR[22] is a program analysis toolkit for Android that takes the app as its input and output an series of xml files, where each xml file estimates the runtime view hierarchy of an activity or a dialog. The estimated runtime view hierarchy is of the same form of static layout files, but may with extra or missing views due to imprecision of static analysis. We present a sample snippet of our adapted GATOR in Section IV-B3. To perform the estimation, GATOR first generates an event flow graph from the event handlers, and then iteratively traverse the graph to add views to activities / dialogs (by scanning Android API methods that add views) until a fix point is reached.

B. FlowDroid

FlowDroid[3] is a novel and highly precise static taint analysis [10] tool for Android applications. A precise model of Android's lifecycle allows the analysis to properly handle callbacks invoked by the Android framework, while context, flow, field, and object-sensitivity allows the analysis to reduce the number of false alarms. For our approach, we configure FlowDroid sources and sinks defined by the SuSi project [20]. SuSi is a tool for the fully automated classification and categorization of Android sources and sinks. In our analysis, the source is often the information, such as unique identifier, location, etc., and the sink can be a network, file, etc. Since the information in an application can flow back and forth, we use SuSi to automatically classify the sources and sinks. After observing the user data flow within the application from sources to sinks, we are able to cognize what data has been collected from the sources such as input control and what data has been sent out to sinks such as dataset servers, Internet, or third party companies. And once we have all the input control APIs, we are be able to detect whether the data flow goes to a sink by applying the FlowDroid.

C. Xposed

The Xposed framework [3] is a tool for the modification of compiled Android apps so they can behave differently at runtime. Xposed takes advantage of the *Zygote* Android daemon, from which all Android apps are forked. By overwriting the process with its own, the framework (and, by extension, its modules) is able to insert hooks into the bytecode of the app allowing the module to perform custom code before and after hooked method calls. Xposed is installed as a normal Android app and runs as a service on the device and thus is able to replace Zygote at boot time. Modules are also written and installed as Android apps. Due to the nature of the framework, Xposed requires root access to the device it is installed upon. The Xposed framework is used in our validation described in Section V-B3.

IV. Approach

The overview of our approach is presented in Figure 5, which consists of two major parts: the manual preparation of domain-specific privacy ontology, and the automatic violation detection given an app and its privacy policy. In the first part, we use the privacy-policy phrases extracted by crowd workers to manually build an ontology that describes semantic relationships between phrases. In the second part, we first apply input context analysis to the app to extract its user input viewsand their context GUI labels. Then, we apply information flow analysis with the extracted user input viewsas data sources to further detect the user input viewswhose received user input flows to network API method invocations, which we referred to as *network-targeted user input views*. Finally, through phrase similarity measurement and pre-constructed phrase-relations in our ontology, we map the context GUI labels of network-targeted user input viewsto the app's privacy policy. If the labels can be mapped to one or more privacy phrases in the ontology, but such phrases are all missing in the policy, we are able to detect a privacy-policy violation. We next introduce the ontology construction in Section IV-A, input

[3] http://repo.xposed.com

context analysis in Section IV-B, and the mapping of GUI labels to policy phrases for violation detection in Section IV-C.

A. Domain Specific Privacy Policy Ontology

To construct the ontology, we perform two main steps: (1) construct the privacy policy lexicon from information types annotated in the privacy policies; and (2) identify semantic relationships between phrases in the privacy policy lexicon. In the ontology, we only consider *hypernyms*, which is an ontological relationships from a more generic phrase to a more specific concept; *meronym*, which is a relationships between a whole and it's parts; and *synonyms*, which is a relationships between two concepts that have nearly same meaning. For example, "credit score" is a part of "credit" and "credit" is subordinate concept to the hypernym"financial information." In addition, the concept indicated by "zip code" is synonymous to "postal code." The ontology can be used for automatic misalignment detection between privacy policies and application code. We now describe how we obtain each lexicon and create the ontology.

1) Extracting the Privacy Policy Lexicon: We used crowdsourcing, content analysis, and natural language processing (NLP) to construct the privacy policy lexicon. First, we selected 5 top applications across four sub-categories (personal budget, banks, personal health, and insurance-pharmacy) in Google Play[4], to yield 10 total apps for the finance and health categories. Next, we segment the privacy policies into 120 word paragraphs using the method described by Breaux and Schaub [6] which yields annotation tasks in each major domains. Figure 6 shows an example annotation task, wherein annotators are asked to annotate phrases based on the following coding frame:

- User Provided Information: any information that the user explicitly provides to the app or other party
- Automatically Collected Information: any information that the app or another party collects or accesses automatically
- Uncertain or Unclear: any information that the app or the other party collect or accesses and it is unclear whether the information is provided by the user or is collected automatically

The user provided information annotations describe types that are explicitly stated in the policies. However, policies do not always mention how or from whom they collect the information. For example, in Figure 6 it is unclear how "personal information" is collected. To build the privacy policy lexicon, we consider both the annotations user provided information, and uncertain or unclear, in case the policy author described the user provided collection in an unclear manner. The remaining code, automatically collected information, ensures that annotators pay close attention how information collection is described in the policy. Among all annotations collected, we only add annotations to the lexicon where two or more annotators agreed on the annotation. This decision follows the empirical analysis of Breaux and Schaub [6], which shows high precision and recall for two or more annotators. In the next step, we applied an entity extractor [5] to the selected annotations to itemize the platform information types into unique entities. Finally, the unique information types are added to the finance or health lexicon depending on which subcategory they belong to.

We recruited nine crowd workers to annotate the 20 privacy policies. In addition to collecting annotations from AMT crowd workers, five of the authors annotated the same privacy policies. We constructed two lexicons in each domain (health and finance) using the annotations from both groups. Comparing the authors lexicon with crowd workers, in the finance domain both groups agreed on 69 unique information types that can be collected though UI input fields that are mentioned in the privacy policies. There are 20 unique information types that the crowd workers annotated which are missed by the authors. Also, there are 23 unique information types that are annotated by authors that are missed by crowd workers. Similarly, in the health domain, crowd workers and authors agreed on 105 unique information types while annotating the policies. Additionally, crowd workers identified 55 unique information types while annotating the policies, while authors identified 34 unique information types that are not annotated by the crowd workers. Comparing the lexicons, the information types annotated by the authors are more fine grained, whereas the information annotated by the crowd workers are more abstract. Furthermore, crowd workers annotated some phrases that cannot be entered by the users in UI input fields. For example, crowd workers annotated "aggregated information" which is not annotated by the authors. To include both crowd workers and authors views, we decided to build the final finance and health lexicons using both crowd workers and authors annotations. In this case, if two or more annotators (crowed workers or authors) agree on an annotation, we add the phrase to the lexicon. Finally, we used the entity extractor developed by Bhatia and Breaux to construct the lexicon with unique information types from the annotated information types [5].

Among 10 policies in finance domain, we constructed 52 HITs with an average word count of 102. These tasks produced 507 annotations including 198 authors annotations and 309 crowd workers annotations. The entity extractor yielded 112 unique information types. Finally, the authors and crowd workers spent 13.05 hours to annotate the finance domain HITs. Similarly, in the health domain, we constructed 141 HITs with an average word count of 105. These tasks produced 1,195 annotations including 456 authors annotations and 739 external annotations. The entity extractor yielded 197 unique information types. Finally, the authors and crowd workers spent 34.65 hours to annotate the health domain HITs. We now discuss how we construct the finance and health ontology using the related lexicons.

2) Constructing the UI-Privacy Policy Ontology: Privacy policy authors tend to use semantic relations to communicate their data practices with users. Among the semantic relations, *hypernymy* is the most common relation used in privacy

[4]https://play.google.com

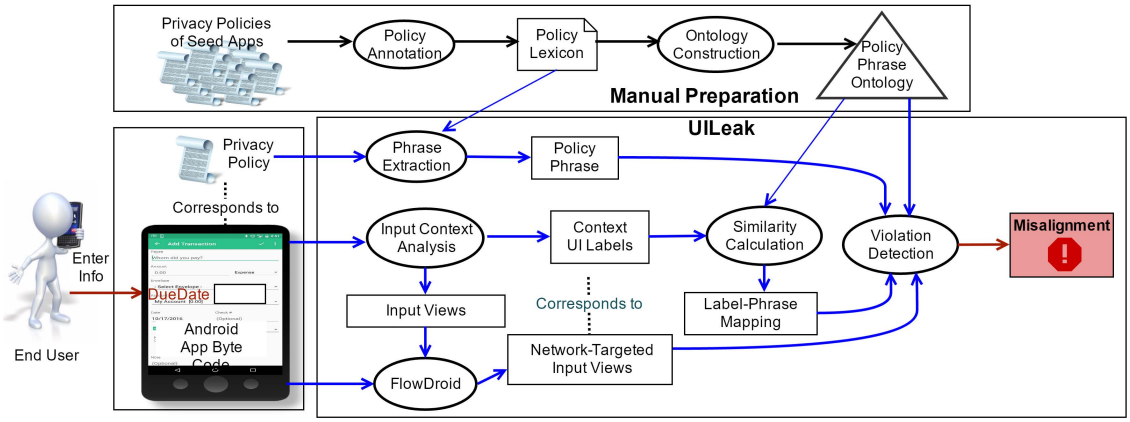
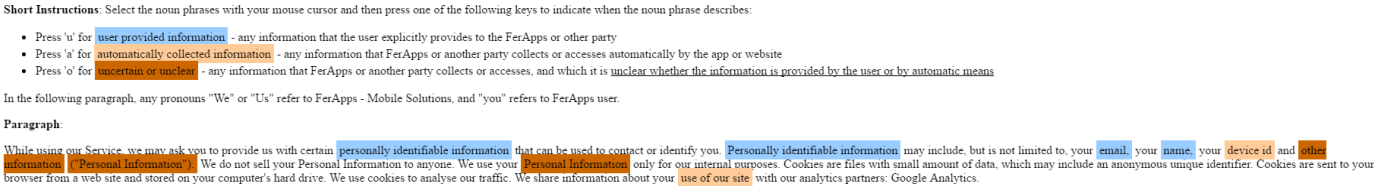


Fig. 5: Approach Overview



Short Instructions: Select the noun phrases with your mouse cursor and then press one of the following keys to indicate when the noun phrase describes:

- Press 'u' for user provided information - any information that the user explicitly provides to the FerApps or other party
- Press 'a' for automatically collected information - any information that FerApps or another party collects or accesses automatically by the app or website
- Press 'o' for uncertain or unclear - any information that FerApps or another party collects or accesses, and which it is unclear whether the information is provided by the user or by automatic means

In the following paragraph, any pronouns "We" or "Us" refer to FerApps - Mobile Solutions, and "you" refers to FerApps user.

Paragraph:

While using our Service, we may ask you to provide us with certain personally identifiable information that can be used to contact or identify you. Personally identifiable information may include, but is not limited to, your email, your name, your device id and other information ("Personal Information"). We do not sell your Personal Information to anyone. We use your Personal Information only for our internal purposes. Cookies are files with small amount of data, which may include an anonymous unique identifier. Cookies are sent to your browser from a web site and stored on your computer's hard drive. We use cookies to analyse our traffic. We share information about your use of our site with our analytics partners: Google Analytics.

Fig. 6: Example of Crowd Sourced Policy Annotation Task

polices where a more general concept is used to describe more specific concepts. For example, the concept "personal information" can be used to convey more specific concepts such as: "credit card information," "medications," among others. Therefore, it is important to identify the semantic relationships between the information types in order to achieve a shared meaning when comparing privacy policies with each other or with application code. To address this issue, we created two ontologies for both finance and health domain using the approach described by Hosseini et al.[12]. The ontologies are used as a reference for identifying the information types not mentioned or mentioned using a more abstract concept in the privacy polices, but collected through UI input fields.

Ontology is a collection of concepts names and relationships between these concepts, including *hypernymy*, *meronymy*, and *synonymy*[12]. Using Description Logic (DL) we define these three relationships as the following axioms in the knowledge base *KB*:

- $AxiomI : C \sqsubseteq D$, (*hypernymy*), which means concept C is a kind of concept D.
- $AxiomII$: C Part Of D, (*meronymy*), which means concept C is a part of concept D.
- $AxiomIII : C \equiv D$, (*synonymy*), which means concept C is equivalent to concept D.

Using our UI input field interpretations $\hat{U}$, we created a mapping from all interpreted tags of an input field to the most frequent tag $U \in \hat{U}$ that represents an input field. Then, we want to infer all policy concepts from policy lexicon $\hat{P}$ $\{P|P \in \hat{P} \wedge KB \models P \sqsubseteq U \vee KB \models P \equiv U\}$. Therefore, we can identify the most frequent tag for each UI input field, then infer the corresponding policy concepts from which at least one concept should be present in the app's privacy policy.

We constructed the ontologies using the manual method introduced by Hosseini et al. [12]. First, we constructed a flat ontology for finance and health lexicon separately, where each phrase in each lexicon is subsumed by the $\top$ concept and no other relationship between phrases exists. Then, for each ontology we follow these steps: (a) we create two copies of the flat ontology $KB1$ and $KB2$ for two analysts; (b) two analysts individually conceptualize the phrases in the flat ontology by comparing each phrase pair, and then we create axioms between concepts from the set of three axioms introduced above; (c) we compare the axioms created by two analysts in $KB1$ and $KB2$ to identify missing axioms and to compute agreement between two knowledge bases. Agreement is measured using the chance-corrected inter-rater reliability agreement statistic Fleiss's Kappa ; (d) finally, two analysts meet to investigate the disagreements and reconcile the axioms in $KB1$ and $KB2$. We again calculate agreement after each reconciliation to measure the improvement from reconciliation.

In the finance domain, the resulting $KB1$ and $KB2$ contain 590 and 582 axioms, respectively. We obtained these results after three rounds of comparisons and reconciliations. The first comparison, yielded 292 differences and after the reconciliation, we were able to reduce the number of disagreements to 43 axioms. We used Fleiss's Kappa to measure the degree of agreement. In the second round of comparison, Kappa was measured 0.83, and after the third round, Kappa increased to 0.92. In the health domain, the resulting $KB1$ and $KB2$ after three rounds of reconciliations contain 951 and 920 axioms, respectively. The first comparison yielded 491 differences and, after the reconciliation, we reduced the number of disagreements to 78 axioms. In the second round of comparison, Kappa was 0.77, and after the third round, Kappa increased to 0.80.

B. Input Context Analysis

In this section, we introduce input context analysis which extract user input viewsand their contextual GUI labels from application code. As mentioned in Section III-A, we leverage Gator to generate a statically estimated UI view hierarchy for each android activity and dialog. However, GATOR has the following 3 limitations that must be addressed to support our application scenario.

First, GATOR does not differentiate input views from other UI views, so we need to identify input views and link them to the API method invocations receiving user input. Second, although GATOR will properly collect and insert the text views holding GUI labels in the generated view hierarchy, it often cannot give the values of the GUI labels because they are generated at runtime through string operations and concatenations. So, we need to further track all the constant strings that are combined to generate GUI labels. Third, as shown in Section II, common dialogs are often used in the apps for receiving user input. Such dialogs are analyzed as separate units in GATOR, but we need to further insert the dialogs back to their parent activities to acquire their context information. We next introduce how we address these limitations.

1) Input View Extraction: The first step in our input context analysis is to extract the input views, or more specifically, the API method invocation that receives user input values from an input view, such as `<android.widget.EditText: android.text.Editable getText()>`. These invocations then serve as the sources of the information flow analysis. We carefully went through the API methods all subclasses of class `View` in android framework, and identified 12 API methods that receiving user inputs, and we list them in our anonymous project site [2]. Note that it is possible that apps acquire information implicitly through navigation events, especially when the information is of enumerated type. For example, while static button labels are not user input, a health app may ask a user to click on either "Male" or "Female" button, and provide different following user interfaces. In our research, we focus on the user input views such as text boxes and check boxes, and we plan to handle the latent user input in future work.

After we collected the user-input-receiving API method invocations, we need to further link them to the view objects in the GUI hierarchy generated by GATOR. Specifically, we insert code to the view object scanning component of GATOR, so that the user-input-receiving API method invocations are added to the view objects as attributes when a view object is scanned by GATOR.

These user-input-receiving API methods are further put into FlowDroid [3] as source APIs. By observing the user data flow within the application from sources to sinks, we are able to cognize whether the data has been collected from the input-receiving API methods to sinks such as dataset sever, internet or third party companies. We use the network sinks in SuSi [20] in our analysis.

2) UI Label Analysis: The second step in our input context analysis is to extract UI labels in the context of an user input view. In this step, we apply the existing string analysis technique [7] to the arguments of all API method invocations that set text to GUI views, such as `<android.widget.Button: void setText( java.lang.CharSequence)>`. Then, we break the value estimation of each argument to a set of strings, so that they can be directly used in our following mapping step. Similarly, we also need to link these set text method invocations to GUI views in the view hierarchy, and we use the same approach as mentioned in input view extraction subsection.

3) Dialog Insertion: Finally, we need to insert the dialogs into their parent activities, and also identify their titles and GUI labels in the context of each parent. Specifically, we search for dialog-showing method invocations (e.g., `<android.app.DialogFragment: void show(...)` in the code and leveraging the points-to analysis results in GATOR to find out the possible types of the dialog (e.g., `GoalSetCheckingInReqDialog`). Then, inside the dialog declaration, we collect all the text setting method invocations in the corresponding builder class, and outside the dialog declaration, we collect all the text setting methods invoked on the dialog object. All the collected texts are added to the dialog object as its attributes, and the dialog itself will be added as an attribute to the view whose event handlers (as collected by GATOR) transitively calls the dialog-showing method invocation.

```
<View type="...TextView" ... title="Set Check-In Requirements">
  <View type="..." ... title="NO_TITLE">
    ...
    <View type="..." idName="et_content" ... getValueOp="[
      <android.widget.EditText: android.text.Editable getText()>
    ]" />
    <View type="..." ... title="Steps"/>
    ...
  </View>
</View>
```

An sample dialog insertion result of adapted GATOR is shown above, where we omitted less-important details for space restrictions. From the example, we can see that, the dialog layout is inserted into the parent activity as a sub view of the `TextView` with title "Set Check-In Requirements".

C. Violation Detection

The primary function of a privacy policy is to inform the user on private information that may be collected by the app. For this reason, we consider violations as *errors of omission* in that the policy failed to notify the user of the particular type of data collection.

We adopt the classifications of violations from existing work by Slavin et al.[24]. Specifically, violations are classified to two major types: *weak* violations where the policy includes vague or abstract terminology to cover the data leak and *strong* violations occurring when the data type is completely omitted from the policy. For example, if the app in question leaks a user's longitude, it would be considered a weak violation if only the phrase "we collect *location* information…" is present in the policy. If no relevant language is included in the policy, it would be considered a strong violation.

In Slavin et. al's work [24], the mapping from API invocations to policy phrases in the ontology is pre-constructed manually. If an API method invocation can mapped to an ontology phrase, but the phrase (or its synonyms in the ontology) does not exist in the privacy policy, a violation is detected. Furthermore, if some of the phrase's hypernyms do exist in the privacy policy, the violation is weak, while if none of the phrase's hypernyms exist in the policy, the violation is strong.

1) Phrase Similarity: In our application scenario, as mentioned in **TC1**, a pre-constructed mapping is not possible. So we consider two well-adopted similarity measurements to map GUI labels to ontology phrases: WordNet similarity [16], and Cosine similarity [23]. Since WordNet calculates similarity only for word pairs, we extend it to map phrase pairs by simple greedy alignment. Specifically, given phrases A and B, we always align the word pairs (one in A and the other in B) with highest similarity, perform the alignment recursively until no more words in either A or B left. For Cosine similarity, we convert the two phrase to two word vectors and apply standard Cosine similarity formula on them.

In our approach, we consider a GUI label and a phrase are mapped if their similarity is higher than a threshold, which is a parameter of our approach, and we study the effectiveness of our approach under different similarity thresholds in our evaluation.

2) Violation Detection Strategy: Unlike API methods which have explicit meanings, the meaning of user input views are implicit and can be understood only from the context of the view. The view id sometimes provide good information, but often not. For example, in our motivation example, the view id of the input box is "et_content", which does not have any specific meaning (so that they can hardly be mapped to the privacy ontology). Furthermore, since the concept used in the user input is infinite, although we restrict our approach to two domains, it is still not possible to exhaust all privacy-related phrases and put them in the ontology. Therefore, if we use the same approach as in Slavin et. al., and directly map the view id / label of a user-input view to ontology phrases, we may miss many violations.

To address this issue, we propose a novel violation detection strategy that takes advantage of the context GUI labels, which we referred to as *Hierarchical Mapping* (and we refer to the original violation detection strategy as *Node Mapping*). Our intuition is that, similar to ontologies, the GUI hierarchy itself conveys information about hypernym relationships. For example, an activity with title "Transaction Information" may contain multiple user input boxes about transaction time, source account, etc, which all are sub-concepts of transaction information. Therefore, when mapping user input views to ontology phrases, we use not only the id / label of the view itself, but also its ancestor ids / labels in the view hierarchy.

In particular, as illustrated in Figure 7, we first collect the labels and ids of all the ancestor views a given input view (light blue views). Furthermore, we collect the labels and ids of views that are sibling views immediate before any collected

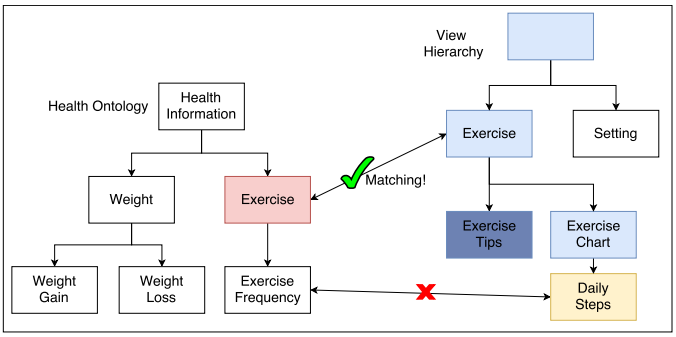


Fig. 7: Illustration of Hierarchical Mapping

ancestor views (dark blue views), because such sibling views often hold text labels of the input views. We refer to the collected ids and labels *ancestor labels*. If an input view's id / label cannot be directly mapped to any ontology nodes, we further map its ancestor labels to the ontology. Note that, the extra mapping may increase the recall but also bring in some noises, but our evaluation shows that the noises are less significant compared with the gain in recall.

V. Evaluation

A. Evaluation Setup

In our evaluation, we collected 100 applications from the Google Play with privacy policies in two categories: finance and health. To make our data set more representative, we further consider two sub-categories in each category. For finance apps, we consider personal-budgeting apps and bank apps, and for health apps, we consider personal-health apps, and medical apps. For each category, the former sub-category is unregulated while the latter sub-category is regulated. Specifically, for each sub-category, we searched the Google Play market with the category name as the query word, and downloaded the highest ranked 25 apps which have privacy policies. For both finance and health categories, we choose the 5 apps with longest privacy policies from each sub-category (in total 10 apps each category) to build our two domain ontologies. We use the rest 80 apps as our evaluation subjects.

B. Ground Truth

One difficult part in our evaluation is to decide ground truth of detected violations. Different users may have different understanding of an input view, and the mapping from UI labels to privacy policies can also be subjective. Finally, the information flows reported by FlowDroid needs to be validated with runtime observation of information leaks to the network. To address these issues, for each violation detected by any variant of our approach, we first use crowd sourcing to elicit the generally acceptable interpretations of the user input views. Then, we followed rigorous steps to map the interpretation of crowd workers to the ontology phrases and privacy polices. Finally, we validate the information flows using Xposed framework. After these 3 steps, in total we collected 20 strong violations and 10 weak violations from 8 apps of the evaluation set.

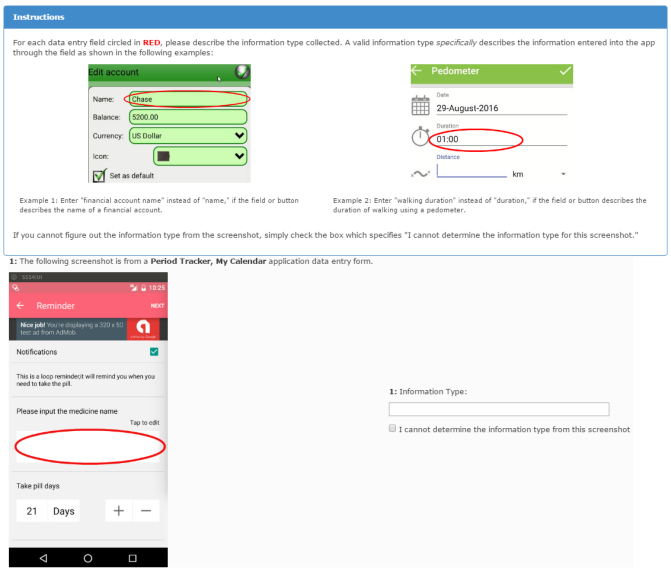


Fig. 8: User Interface Input Field Tagging Task

1) Eliciting UI Input Fields Interpretations: Inferring the information types from the text field type in the application code alone is a challenging task. We analyzed 37 input fields names and found only 29.7% percent correctly describe the field type. Therefore, in this section we describe the approach we used to extract the information types associated with each input field. We designed a free listing survey (defined in Bernard et al.'s work [4]) that asks individuals to identify the information type that describes the information entered into the app through a specific UI input field shown in a screenshot. Each survey consisted of 3-5 screenshots, and Figure 8 presents an example screenshot. We surveyed 37 input fields from nine apps that collect information types through UI input fields without mentioning in their privacy policies. We recruited 30 participants per survey using Amazon Mechanical Turk concluding 242 Human Intelligence Tasks (HITs). Participants of the surveys were located in the United States with HIT approval rate greater than 95%.

For each input field, we obtained 30 information types, which we now call tags. Because there are multiple ways to describe the same concept, we pre-processed the results to more easily compare tags as follows: (a) rewrite prepositional phrases into noun phrases, e.g. "amount of money" is rewritten to "money amount;" (b) remove possessives, e.g., "user's current medication" is changed to "user current medication;" (c) replacing "your" with "user", e.g., "setting your own pace" is changed to "setting user own pace;" (d) remove hyphens, e.g., "e-mail" is changed to "email." This step is similar in purpose to porter stemming in natural language processing [18], [19]. After pre-processing, we combine similar tags for each field and calculate the tag frequency. Finally, each field is represented by the most frequent tag that is also linked to a tag set that contains the least frequent tags for that field. That tag set can be used to expand the interpretation of information types for the same input field, and also can be used to map the UI input field to the privacy policy lexicon which is described later in section V-B2.

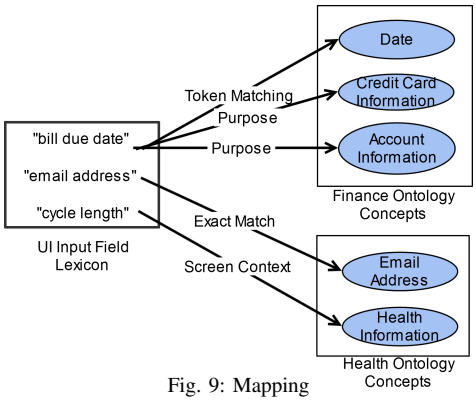


Fig. 9: Mapping

2) Mapping: We need to further map elicited descriptive phases to the ontology and the privacy policy. We follow the three step approach: (1) for each elicited phrase, we look for the exact match of the phrase in the ontology. If the match is found, we map the phrase to the matched concept in the ontology; (2) if we cannot find the exact match, (a) we break the phrase into atomic tokens and create the superset using the atomic tokens and combinations of them. We try to find the exact match of the elements in the superset with the concepts in the ontology. If we find an exact match, we map the phrase to the matching concept in the ontology; (b) in this step, we identify the purpose for the UI input field phrase using the existing concepts in the ontology. Then, we map the phrase to the concept that presents the purpose of the phrase. (3) if we weren't able to find related concepts for the phrase using the previous steps, we use the context of the screen where the input field is presented in the application. The context provides us guides to find related concepts to the phrase in the ontology. Figure 9 shows the mapping for some phrases in UI input field lexicon. For example, for phrase "bill due date", we follow the two step approach to map this phrase to some concepts in the ontology. In step 1, we look for the exact match of "bill due date" in the finance ontology and we fail to find the match. Therefore, we try to find matching concepts using step 2. First, we tokenize the phrase $S = \{bill, due, date\}$. Next, we create the superset of S where $T \supset S$. Therefore, $T = \{bill, due, date, billdue, billdate, duedate, billduedate\}$. Comparing the elements of set T with the concepts in the finance ontology, we are able to find a match for "date." Finally, we try to find a purpose for "bill due date" using the concepts in finance ontology. We found "account information" and "credit card information" as two concepts that can be associated with "bill due date." In another example, we were not able to find the associated concepts to "cycle length" using the two first steps. Therefore, using the context of the screen that contains the UI input field and the application itself, we were able to identify "cycle length" is related to "menstrual information" and not "exercise information." Therefore, we mapped "cycle length" to "health information." Note that we have the authors but not crowd workers to perform this mapping because it requires much expertise,

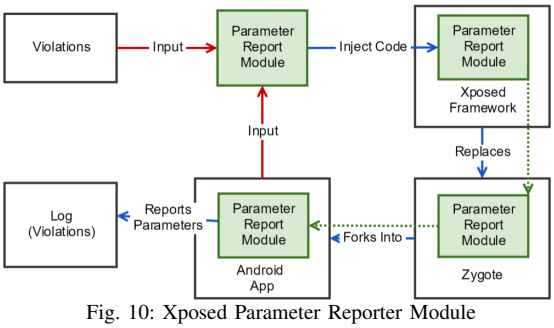


Fig. 10: Xposed Parameter Reporter Module

and a voting result is used for ground truth. For mapping the elicited phrase to the privacy policy, we read the policies and use our intuition to do the mapping because the policy is free-style. Similarly we use the voting results among 5 of the authors for ground truth.

3) Validation with Xposed: In order to verify the information being passed to network sinks, we implemented runtime tool to "hijack" the apps we tested as they ran on Android and report the input parameters to the sinks involved in the suspected leaks. To do so, we created a module that utilized the Xposed framework. The approach is depicted in Figure 10.

Our verification module, the Parameter Reporter, searches for the specific sinks used in the violations. For example, we detected the `org.apache.http.client.HttpGet()` sink to leak information in the `cc.pacer.androidapp` app so the module was written to detect calls to that method from that app.

The integration of the module with the app at run time can be seen in the figure starting from the top right box. The module is loaded by the Xposed framework at boot time and thus is transferred into Zygote (which Xposed replaces). When the app is started, the process is forked from Zygote and the module persists within the app's bytecode along with the hooks included for detection of the sinks.

The custom code written in our module simply writes the input parameters for the sinks to a log file. This allows us to trigger the leak of data at runtime and verify that the UI input values were leaked. For example, if we detected a leak from a text input to `HttpGet()`, we can enter an easily-recognizable string to that text input, "LEAKED DATA", trigger the leak at runtime, and check the log file to see if "LEAKED DATA" was passed to `HttpGet()`.

C. Violation Detection Results

After the ground truth violations are confirmed, we can compare different variants of our approach on their effectiveness. Since we have the similarity threshold as a parameter, we also want to observe how the effectiveness change as similarity threshold changes. We use the *precision*, *recall* and the *F-score* as our metrics. Note that, since it is impossible to detect all violations in the evaluation set, our *recall* is actually *relative recall*, which uses the detected violations among all technique variants as the whole set of violations.

In our experiment, we consider two similarity measurements: WordNet and Cosine. We also consider two violation detection strategies: node-mapping where only label / id of the user input view is considered, and hierarchical-mapping where all ancestor labels / ids are considered. So we have four variants of our approach by combing the techniques: Hier+WN, Hier+Cos, Node+WN, Node+Cos.

The violation detection results of our approach is presented in Figure 11. In each sub-figure, we compare the four variants on different similarity thresholds (0-1), with the legend on the right top corner of the chart. The figures in Row 1 are comparing the precision, recall, and F-score on strong violations, respectively. The figures in Row 2 are comparing the precision, recall, and F-score on violations with strict types, respectively. Here, by strict types, we mean a violation is considered correctly detected only if the type is also correct (strong violations detected as strong, and weak violations detected as weak). The figures in Row 3 are comparing the precision, recall, and F-score on violations with general types, respectively. By general types, we mean a violation is considered correctly detected even if the type is wrong (strong violations detected as weak, and weak violations detected as strong).

From the figure, we have the following observations. First, hierarchical-mapping variants (solid lines) performs much better (averagely 20 percentage points in recall, and 13 percentage points in F-score) than node-mapping variants in both recall and F-score, while the precision of the two techniques are similar. This observation confirms our intuition that the incorporation of context GUI labels can greatly improve the violation detection results (note that recall is more important than precision in this scenario as long as the precision difference is not too large).

Second, as similarity threshold increases, the effectiveness of WordNet-based variants climbs up, while the effectiveness of Cosine-based variants drops. With the similarity threshold around 0.8, all variants are close to their best performance (F-score). The reason is that, WordNet often gives high similarity scores to words that are not close related in the domain (but may be closely related in other domains, such as bank and river). So increasing similarity threshold helps remove noises. By contrast, since Cosine similarity requires exact word match, too high similarity threshold will result in losing matches between GUI labels and ontology phrases.

Third, with 0.8 similarity threshold, our hierarchical-mapping-based variants can achieve 60% F-score and 65% recall for strong violations, 53% F-score and recall for strict type violation detection, and 84% F-score and 86% recall for general violation detection.

Fourth, as the similarity thresholds change, precision and recall of a variant sometimes change in a similar way. The reason is that, as the similarity threshold changes, there will be more (or less) mappings between GUI labels and ontology phrases. The additional mappings may help catch violations when a privacy-related GUI label was not mapped to any phrases, but may also hide violations when GUI labels are mapped to more ontology phrases which may appear in the privacy policies.

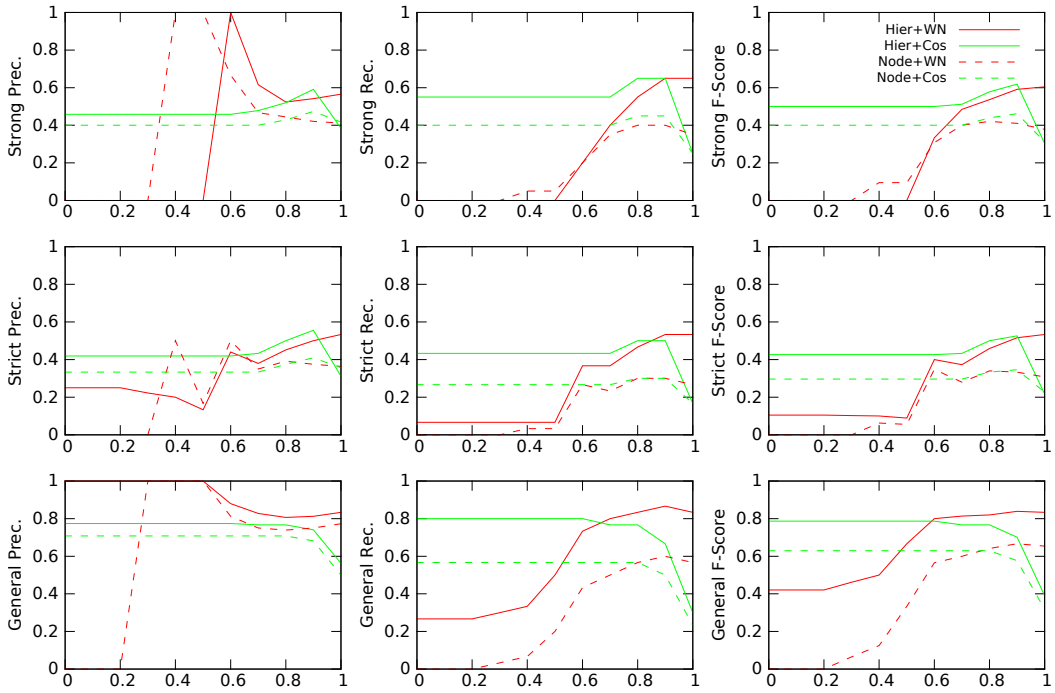


Fig. 11: Comparison of Technique Variants under Different Similarity Thresholds

Examples. We hereby describe some real examples about privacy violations. To avoid legal issues, we do not reveal the name of any apps described. One example of strong violation was found in one of the top pregnancy related health apps with more than 50 million installs. We found that the app send information about cycles and medicines taken to the servers, but it does not mention anything about gathering such information. By contrast, a weak violation was found in a top personal budgeting app with more than 1 million installs. We found that the app sends the bill due date information to the server, but it is not directly mentioned in the privacy policy, although transactions as a higher-level phrase is mentioned.

D. Threats to Validity

The major threats to our internal validity is the human mistakes and subjectiveness in our implementation of the approach and the labeling of the ground truth. To reduce this threat, we use existing frameworks (e.g., FlowDroid and GATOR) in our implementation, and leverage crowd workers in labeling to reduce subjectiveness. The major threats to the external validity is that our findings may apply to only our evaluation set or the two domains we are working on. To reduce this threat, we collected top apps from different categories, and we work on the two domains which are especially important for privacy. We also plan to apply our approach to more domains and apps.

VI. Discussion

Our privacy policy ontology and violation detection techniques are sufficiently general to work well with dynamic analysis, such as monitoring. The static analysis results can be leveraged to conduct a more efficient dynamic analysis by leveraging the static analysis results to focusing on some targeted methods. For example, if static analysis shows utilization of the method `foo()` to access or transmit sensitive data, the method described in Section V-B3 can be used to detect invocations of `foo()` as well as any sensitive parameters passed to it at runtime. This potentially allows for communication to the end user of violations in real time.

Our approach can be utilized to generate privacy policy statements from the mobile app source code. It will begin with information flow descriptions which includes a set of UI-related information flows. Then, the flow summarization and flow prioritization techniques need to be developed to generate policy statements, such as determining the appropriate level of abstraction of concepts within the ontology to use. It would be appropriate to generate actual privacy policy statements in a controlled natural language that can be included in an apps privacy policy.

The severity of a weak violation (i.e., the level of abstraction of the term in the policy) can be further refined by applying semantic distance [25] to the represented term and the mapped term. Currently, a violation where the policy includes a hypernym for the mapped term, regardless of how abstract it is, is categorized as weak leading to a large number of weak violations. By quantifying the level of abstraction, vague representations (e.g., those that do not contain strong qualifying adjectives) could be identified as instances where intentional obfuscation may be present. For example, if a privacy policy states that the *personal information* provided by end users is collected without presenting any specific type of personal information, the *bill due date* collected by an app code will be deemed as a stronger violation than the case

where its policy states that *financial information* is collected. Similarly, if the label of the view, the title of the UI page, or the instructions do not directly contain any ontology concept, we can also categorize it as a stronger violation.

VII. Related Work

Prior work exists on the exploration of information flow graphs between different views, observation of hidden data flows from given APIs sinks, and the detection of potential privacy policy violations in Android app code. To our knowledge, ours is the first approach that uses data flow analysis to verify consistency between app-collected data and privacy policy language with regard to native code and user input. The following are related works in the area of Android data flow analysis and privacy policies.

Slavin et al. used a similar approach to detect privacy policy violations in Android apps based on Android API calls [24]. Such an approach is useful in identifying leaks where a device's sensors (e.g., GPS, Bluetooth, WiFi, etc) produce sensitive information. The authors used a similar policy phrase ontology as well as an API to phrase mapping for identifying weak and strong violations. The major difference our approach has is the ability to detect violations based on native code. By mapping privacy-policy Phrases to user input views, we are able to go beyond Android API-based detection and identify potential violations involving user input. Furthermore, the API-based approach relies on developer documentation for the mapping whereas policies are not typically written by developers. For our approach, privacy-policy Phrases are mapped with a user-oriented perspective which is closer to the language of privacy policies, which we assert is more relevant since privacy policies are written with an intent to be understood by end users.

Existing work on UI-related information explore various facets of privacy and security. Huang et al. match text from UI components to top-level functions in order to detect clandestine behavior [15]. Their work targets functions by identifying suspicious permissions. In contrast, our work compared the consistency between privacy policies and user input via UI components which are based on native code. This allows our approach to not be limited by the coarse granularity of Android permissions.

Huang et al. [15] modeled so-called "stealthy behaviors" as program behavior that mismatches with user interface by using static analysis. They extracted text from the user interface component and match them with top-level functions. Similarly, SUPOR [13] and UIPicker [17] identify sensitive input views to which sensitive information can be entered. These techniques are provide the means to identify critical data entry points however, they lack the necessary NLP components for relating such behavior to privacy policies.

Huang et al. developed BIDTEXT as a tool for reporting the propagation of label set variables corresponding to sensitive text labels to sinks [14]. BIDDTEXT is not able to solve our problem because it does not try to map sensitive labels to phrases in the privacy policies.

VIII. Conclusion

In this paper, we proposed a novel approach to detect privacy policy violations due to leak of user input data. To addressed the two technical challenges (infinite mapping and various GUI implementation), we adapted the GATOR framework, and developed hierarchical-mapping-based violation detection. We apply our approach on two important domains (finance and health) and detected 20 strong violations and 10 weak violations in 80 top apps from the domains. Our experiment shows that our best technique variant can achieve a F score of 84% with proper similarity threshold set.

References

[1] Google play statistics, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed: 2017-02-22.

[2] Ui privacy project web site, https://sites.google.com/site/uiprivacy2017/. Accessed: 2017-02-22.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, Jun 2014.

[4] H. R. Bernard. *Research methods in anthropology: Qualitative and quantitative approaches*. Rowman Altamira, 2011.

[5] J. Bhatia and T. D. Breaux. Towards an information type lexicon for privacy policies. In *Requirements Engineering and Law (RELAW), 2015 IEEE Eighth International Workshop on*, pages 19–24. IEEE, 2015.

[6] T. D. Breaux and F. Schaub. Scaling requirements extraction to the crowd: Experiments with privacy policies. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 163–172. IEEE, 2014.

[7] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[8] S. B. Committee. Gramm-leach-bliley act, 1999. Public Law 106-102.

[9] P. S. S. Council. Payment card industry data security standard, 2016. version 3.2.

[10] C. F. (ec Spride, S. A. (ec Spride, S. R. (ec Spride, E. B. (ec Spride, A. Bartel, C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, R. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. Mcdaniel. Damien octeau (penn state university) patrick mcdaniel (penn state university) highly precise taint analysis for android application, 2013.

[11] F. A. Henry Reuss. Right to financial privacy act, 1978. Public Law 95-630.

[12] M. B. Hosseini, S. Wadkar, T. D. Breaux, and J. Niu. Lexical similarity of information type hypernyms, meronyms and synonyms in privacy policies. In *2016 AAAI Fall Symposium Series*, 2016.

[13] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 977–992, Berkeley, CA, USA, 2015. USENIX Association.

[14] J. Huang, X. Zhang, and L. Tan. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 169–180, New York, NY, USA, 2016. ACM.

[15] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

[16] A. Kilgarriff and C. Fellbaum. Wordnet: An electronic lexical database, 2000.

[17] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 993–1008, Berkeley, CA, USA, 2015. USENIX Association.

[18] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[19] M. F. Porter. Snowball: A language for stemming algorithms, 2001.

[20] S. Rasthofer, S. Arzt, E. Spride, T. U. Darmstadt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks, Feb 2014.

[21] H. Resources and S. Administration. Health insurance portability and accountability act, 1996. Public Law 104-191.

[22] A. Rountev and D. Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 143:143–143:153, New York, NY, USA, 2014. ACM.

[23] G. Salton, A. Wong, and C.-S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[24] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 25–36, New York, NY, USA, 2016. ACM.

[25] Z. Wu and M. Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics, 1994.