GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications

Xiaoyin Wang
Xue Qin
University of Texas at San Antonio
{xiaoyin.wang, xue.qin}@utsa.edu

Mitra Bokaei Hosseini
Rocky Slavin
University of Texas at San Antonio
{mitra.bokaeihosseini, rocky.slavin}@utsa.edu

Travis D. Breaux
Carnegie Mellon University
breaux@cs.cmu.edu

Jianwei Niu
University of Texas at San Antonio
jianwei.niu@utsa.edu

ABSTRACT

The Android mobile platform supports billions of devices across more than 190 countries around the world. This popularity coupled with user data collection by Android apps has made privacy protection a well-known challenge in the Android ecosystem. In practice, app producers provide privacy policies disclosing what information is collected and processed by the app. However, it is difficult to trace such claims to the corresponding app code to verify whether the implementation is consistent with the policy. Existing approaches for privacy policy alignment focus on information directly accessed through the Android platform (e.g., location and device ID), but are unable to handle user input, a major source of private information. In this paper, we propose a novel approach that automatically detects privacy leaks of user-entered data for a given Android app and determines whether such leakage may violate the app's privacy policy claims. For evaluation, we applied our approach to 120 popular apps from three privacy-relevant app categories: finance, health, and dating. The results show that our approach was able to detect 21 strong violations and 18 weak violations from the studied apps.

CCS CONCEPTS

•Software and its engineering → Software verification and validation;

KEYWORDS

Mobile privacy policy, Android application, User input

ACM Reference format:
Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D. Breaux, and Jianwei Niu. 2018. GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.
DOI: 10.1145/3180155.3180196

1 INTRODUCTION

Mobile applications (apps) are becoming increasingly pervasive. By June 2017, the Google Play Store surpassed three million apps [1], and the Android platform held 85% of the smartphone OS market share [2]. Among these apps, some categories of apps can be particularly privacy-sensitive. In 2015, health apps were downloaded by 58% of mobile phone users [17]. Such apps collect information on body measurements, diet, exercise, and medical treatment, among others. Similarly, 73% of the personal finance app Mint's 20 million users pay their balances every month through the app [3].

With ease of access to personal information and the large scale of mobile app deployment, developers need better tools to help protect user privacy. Google encourages app developers to provide users with privacy policies [26], however, innovation and competition among mobile app developers can introduce inconsistencies between the application code and privacy policies. Unlike security threats where malicious developers hoard personal data, the privacy threat motivating our work is the developer who unintentionally collects personal data without informing the policy author, or where the software changes over time to yield and outdated policy.

Prior work by Slavin et al. [26] and Zimmeck et al. [33] traced privacy policy statements about the collection of platform information to application program interface (API) calls using static program analysis. These API calls concern personal information that is automatically collected from the device, such as user location, device identifiers, contact information, and sensor data. This prior work is limited, because it does not account for personal data that *users provide directly through an app's graphical user interface (GUI).* Figure 1 shows an example where sensitive data is provided to the app via the interface and is thus disconnected from any API call. In Figure 1, the user manually enters the steps they have taken using a text field. There are many ways, both static and dynamic, to render the field and link the information provided through the field to program-level data types. Furthermore, the field itself may not have tight constraints on the input values, making it difficult to determine the information type.

These unaddressed, GUI-related challenges further widen the gap between privacy policies and app-based data practices. To bridge this gap, we identified two new technical challenges that we address in this paper:

TC1: Vague and Unbounded Information Types for User Input Data. The information types automatically collected through

ICSE '18, Gothenburg, Sweden
© 2018 ACM. 978-1-4503-5638-1/18/05...$15.00
DOI: 10.1145/3180155.3180196

platform API methods are constrained to what is collectible by hardware commonly shared across mobile devices. This constraint limits the terminological space to only a few general category names (e.g., location, voice, camera). 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 and language types.

TC2: Varying GUI Implementation Techniques. Unlike platform API method calls that can be detected by scanning the app byte code, user interfaces can be implemented using static declarations in resource files or programmatically in the code. Techniques, such as SUPOR [13] and UIPicker [18], can be used to identify input views (GUI views accepting user inputs) receiving sensitive user input, but they do not map these views to relevant policy terms, nor do they identify programmatically-generated input views.

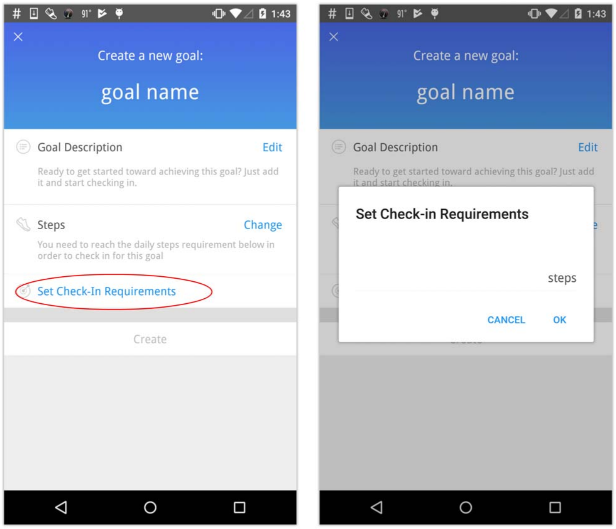
In this paper, we present a novel technique to detect privacy-policy violations on user-provided information for Android apps. The approach maps each input view to policy terminology through an ontology, and then performs static information flow analysis to detect information flows that violate relevant policy statements. To address **TC1**, for each input view in a new app, we use phrase similarity measurements to map GUI labels together with its context to ontology concepts. The ontology is then matched to the policy text. To address **TC2**, we developed a GUI analysis technique to estimate the structure of programmatically generated GUIs and collect all GUI labels in the context of a given input view. Our analysis is based on GATOR [24], an existing GUI analysis framework for Android.

To validate our approach, we focus on three app categories in the Google Play Store: health, finance, and dating. These three categories are important because they can require access to sensitive personal information. Furthermore, the first two domains are regulated by the Health Insurance Portability and Accountability Ac (HIPAA) [23] and Gramm-Leach-Bliley Ac (GLBA) [11], respectively. In our experiment, we collected 150 of the most popular apps and their privacy policies (50 apps for each category), setting aside 20% of the apps for training. Using the privacy policies from the training apps, we constructed an ontology for each of the three domains. We then applied our approach to the remaining 120 apps and detected 39 violations, which we manually confirmed by recording the runtime network requests with the Xposed framework [5].

The contributions are as follows.

- We developed a novel GUI-analysis approach to detect inconsistencies between the app's code, collection behavior of user input data and collection statements in mobile app privacy policies.
- Using crowd sourcing tools, we developed domain-specific privacy ontologies for three privacy-sensitive domains: health, finance, and dating.
- We experimentally evaluated the approach on 120 most popular apps in the health, finance, and dating and detected 21 strong violations and 18 weak violations.

This paper is organized as follows: Section 2 presents motivation and an example; Section 3, describes our framework and approach; Section 4 describes the evaluation, with the discussion in Section 5; Section 6 includes related work; and we conclude with future work in Section 7.



Ready to get started toward achieving this goal? Just add it and start checking in.

You need to reach the daily steps requirement below in order to check in for this goal

Figure 1: Screenshots from Pacer

2 MOTIVATING EXAMPLE

In this section, we demonstrate the difficulties of understanding input views due to their varying implementations.

```
<LinearLayout android:gravity="center_horizontak" ...>
 <cc.pacer.androidapp.ui.common.fonts.TypefaceTextView
  ... android:id="@id/title" ... />
 <cc.pacer.androidapp.ui.common.fonts.TypefacedEditText
  ... android:id="@id/et_content" ... />
 <LinearLayout ...>
  <Button ...
   android:id="@id/btnLeft" ...
   android:text="@string/btn_cancel" ... />
 </LinearLayout>
</LinearLayout>
```

Listing 1: Partial Code from common_input_dialog.xml

```
const v2, 0x7f07011a
invoke-virtual {v1,v2}, Lcom/afollestad/materialdialogs/
      MaterialDialog\$Builder;->title(I)Lcom/afollestad/
      materialdialogs/MaterialDialog\$Builder;
move-result-object v1
```

Listing 2: Virtual Invoke Example in GoalSetCheckingInReqDialog.smali

```
<LinearLayout android:gravity="center_horizontak" ...>
<cc.pacer.androidapp.ui.common.fonts.TypefaceTextView ...
 android:text="@string/goal_set_requirements" ... />
```

Listing 3: Partial Code from goal_create_details_fragment.xml

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

The example in Figure 1 shows two GUI screenshots from Pacer, a popular fitness app, depicting the GUI when a 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 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 and dynamic layouts. Listing 1 shows the static definition of the right screenshot, which is a *layout template* defining the basic layout structure and the font / style information. All the labels and IDs are vaguely defined (e.g., Dialog title is undefined, and "et_content" is used as the view id of the input box). 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.

Listing 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 at Line 1 as `v2` with the id `0x7f07011a`. Here, the id references the appropriate string in `string.xml` based on the context. In Line 2, `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 difficult to understand what is supposed to be entered into the field. 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 Listing 3, and the view's label referring to `@String/goal_set_requirements`.

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 difficult. In this 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.

3 APPROACH

The overview of GUILeak is depicted as a data flow diagram (see Figure 2), which consists of three stages: (a) the ontology construction stage (blue) creates a baseline ontology by extracting reusable concepts from multiple privacy policies; (b) the app policy tracing stage (green) yields phrases that describe data types that are collected automatically or from the user directly based on a target app's privacy policy; and (c) the GUI analysis stage (orange) that extracts the input fields, field labels, and view identifiers from the input views, which are then fed to data flow analysis (i.e., FlowDroid [6] is used in GUILeak) as information sources.

The mappings are generated from the ontology, the data type phrases, and the input view IDs and labels, and are used to detect policy violations in the identified input data flows. The mappings allow us to detect two kinds of gaps: (1) when the policy is too abstract (e.g., it refers to "personal information" when the app shares your age and weight), and (2) when data types are collected and shared but they are not described in the policy. In this paper, we are specifically interested in network-targeted input flows in which user provided information flows to any network API method invocation.

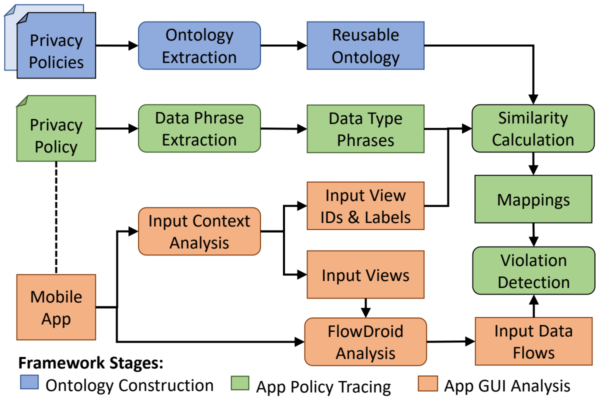


Figure 2: Approach Overview

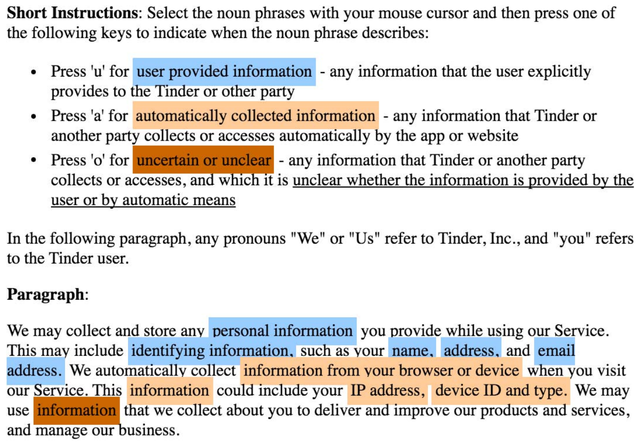
We introduce the specific steps in the ontology construction in Section 3.1, the input context analysis in Section 3.2, and the mapping of policy phrases to GUI labels for violation detection in Section 3.3.

3.1 Ontology Construction

To construct the ontology, we perform two main steps: (1) construct a privacy policy lexicon from information types extracted from the privacy policies; and (2) identify semantic relationships among phrases in the privacy policy lexicon to yield the ontology. The ontology models three semantic relationships: the *hypernym*, which is an ontological relationship from a more generic concept to a more specific concept; the *meronym*, which is a relationship between a whole and its parts; and the *synonym*, which is a relationship between two concepts with nearly the same meaning. The ontology can be used for automatic violation detection between privacy policies and application code.

3.1.1 Extracting Information Type Phrases. The information type phrase extraction step combines crowdsourcing, content analysis, and natural language processing (NLP) to construct the privacy policy lexicon. For our study, we first select five top applications in each of six sub-categories (personal budget, banks, personal health, insurance-pharmacy, casual and serious dating) in Google Play, to yield 30 total apps for the finance and health categories. Next, we segment the privacy policies into 120 word paragraphs using the method described in [9], which yields annotation tasks from each policy. Figure 3 shows an example annotation task, wherein annotators are asked to annotate phrases based on the following coding frame: User-Provided Information; Automatically Collected Information; and Uncertain or Unclear.

The user-provided information annotations describe types explicitly stated in the policies. However, policies do not always mention how or from whom they collect the information. For example, in Figure 3, it is unclear how "information" is collected. To build the privacy policy lexicon, we consider both annotations coded as user-provided information, and uncertain or unclear, in case the policy author described the user-provided collection in an unclear manner. We collect annotations by recruiting five crowd workers from Amazon Mechanical Turk (AMT) to annotate each 120-word



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 Tinder or other party
- Press 'a' for automatically collected information - any information that Tinder or another party collects or accesses automatically by the app or website
- Press 'o' for uncertain or unclear - any information that Tinder 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 Tinder, Inc., and "you" refers to the Tinder user.

Paragraph:

We may collect and store any personal information you provide while using our Service. This may include identifying information, such as your name, address, and email address. We automatically collect information from your browser or device when you visit our Service. This information could include your IP address, device ID and type. We may use information that we collect about you to deliver and improve our products and services, and manage our business.

Figure 3: Example of Crowd Sourced Policy Annotation Task

paragraph of the combined 30 privacy policies. Because this annotation task differed from [9], we also collected annotations for the same tasks from the authors to evaluate the crowd worker lexicon. Among all annotations collected, we only add annotations to the lexicon where two or more annotators agreed on the annotation. This decision follows the study which shows high precision and recall for two or more annotators [9]. In the next step, we applied an entity extractor [8] to the selected annotations to itemize the information types into unique entities. Finally, the unique information types are added to the finance, health, or dating lexicon depending on which sub-category they belong to.

3.1.2 Identifying Semantic Relationships. *Hypernymy* is the most common relation found in privacy polices. For example, the concept "personal information" can be used to generalize more specific concepts, such as: "credit card information" or "medications." Therefore, it is important to identify the semantic relationships between the information types found in policies to account for how policies can generally refer to a code-level collection of a more specific information type, as opposed to when those policies omit any mention of the collection practice. To address this issue, we constructed separate ontologies from the finance, health, and dating lexicons. These lexicons share a small number of phrases supporting our decision on constructing individual ontologies for each domain. The ontologies are constructed using the heuristics below [12]:

- Hypernym: $C \sqsubseteq D$, which means concept D is a general category of C, e.g., "password" is a kind of "authentication information."
- Meronym: C Part Of D, which means concept C is a part of concept D, e.g., "email message" is a part of "email."
- Modifiers: $C_1\_C_2 \sqsubseteq C_2$ and, $C_1\_C_2 \sqsubseteq C_1\_information$, which means concept C_1 is modifying concept C_2, e.g., "mobile phone number" is a kind of "phone number" and "mobile information."
- Plural: $C \equiv D$, which means concept C is a plural form of concept D, e.g., "addresses" is equivalent to "address."
- Synonym: $C \equiv D$, which means concept C is a synonym of concept D, e.g., "geo-location" is equivalent to "geographic location."
- Thing: $C_1 \equiv C_1\_information$, when concept C has logical boundaries and can be composed of other concepts, e.g., "name" is equivalent to "name information."
- Event: $C_1 \equiv C_1\_information$, when concept C describes an event, e.g., "usage" is equivalent to "usage information."

Semantic relationship identification begins with an ontology, wherein each lexicon phrase is subsumed by the $\top$ concept and no other relationship exists between phrases. Next, two analysts follow four steps (see [12]): (1) they create two copies of the initial ontology $KB1$ and $KB2$, one for each analyst; (2) each analyst individually compares each phrase pair, and creates hypernymy, meronymy, or synonymy axiom between concepts when an appropriate relationship is found based on the above heuristics; (3) the two analysts compare their axioms in $KB1$ and $KB2$ to identify missing axioms and to compute the degree of agreement. Agreement is measured using the chance-corrected inter-rater reliability statistic Fleiss's Kappa; and (4) finally, two analysts meet to investigate the disagreements and reconcile the axioms in $KB1$ and $KB2$. The analysts re-calculate agreement after each reconciliation to measure the improvement due to reconciliation. Identifying semantic relationships is a heuristic-based procedure, wherein each analyst develops their own heuristics or rules for identifying relationships. Once all disagreements are reconciled, the two KBs are equivalent and each one can be used in GUILeak.

3.2 Input Context Analysis

The input context analysis stage serves to extract user input views and their contextual GUI labels from the app code. In GUILeak, we adapt GATOR [24] to generate a statically-estimated GUI view hierarchy, which includes the input identifier, layout, and form elements for each Android activity and dialog. GATOR is a program analysis toolkit for Android that takes the app as its input and produces an estimated XML hierarchy of activities and dialogs. GATOR first generates an event flow graph from code and then iteratively traverses the graph to add views to activities / dialogs (by scanning Android API methods that add views) until a fix point is reached.

For our goal, GATOR has three limitations. First, GATOR does not distinguish between input views and other GUI views, so we need to identify and link input views to the API method invocations receiving user input. Second, although GATOR properly collects and inserts the text views holding GUI labels in the generated hierarchy, it often cannot provide the GUI label values because they are generated at runtime. Thus, we must trace string values back to the string constants defined elsewhere in the code. Third, as shown in Section 2, common dialogs can be used to receive user input. While GATOR analyzes these dialogs as separate units, they must be linked back to their parent activities so more context information can be extracted. The resulting GUI view hierarchies can represent the complete input context. We next introduce how we address these limitations.

3.2.1 Input View Extraction. The first input context analysis step is to extract the API method invocation that receives user input from an input view, such as `<android.widget.EditText: android.text.Editable getText()>`. These invocations serve as the *sources* in the following information flow analysis. To support this extraction, we carefully reviewed all API methods in subclasses of the Android framework `View` class, and identified 12 API methods that receive user inputs. The list of these methods is available in our anonymized project website [4]. It is also possible that apps acquire user input implicitly through navigation events, especially when the input is an enumerated type. For example, while static button labels are not user input, a health app may ask a user to click on either a "Male" or "Female" button, which leads to different subsequent user interfaces. In our research, we focus on the user input views such as text boxes and check boxes, and plan to extend the approach in future to include latent user input.

Next, we insert the user-input-receiving API methods into the view objects contained in the GATOR-generated GUI view hierarchy. This is achieved by inserting code into 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 configured into FlowDroid [6] as source API methods. By observing the user data flow within the application from sources to sinks, we can recognize whether the data has been collected from the input-receiving API methods and shared with remote sinks on the Internet. In our analysis, we use the network sinks in SUSI [22].

3.2.2 Input Label Analysis. The second input context analysis step is to extract GUI labels in the context of an user input view. In this step, we apply existing string analysis technique [10] 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 into a set of strings, so that they can be directly used in the subsequent mapping step. We also link the `setText()` method invocations to GUI views in the view hierarchy using the same approach mentioned in Section 3.2.1

3.2.3 Dialog Insertion. Finally, we insert the dialogs into their parent activities, which allows us to identify the dialog titles and GUI labels in the context of each parent. Specifically, we scan the code for dialog-showing method invocations (e.g., `<android.app.DialogFragment: void show(...)>`) and leverage the points-to analysis results from GATOR to discover the dialog types (e.g., `GoalSetCheckingInReqDialog`). Next, 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. The collected text-setting method invocations are added to the dialog object as attributes. Then GUILeak acquires possible arguments of the text-setting methods with input label analysis, and add them to the view hierarchy of the dialog. Finally, the dialog itself is added as an attribute to the view whose event handlers (identified by GATOR) transitively call the dialog-showing method invocation.

In Listing 4, we show a sample dialog insertion result from the extended GATOR; minor details were omitted for space. In the example, we see that the dialog layout was inserted into the parent activity as a sub view of the `TextView` with title "Set Check-In Requirements."

3.3 Mapping and Violation Detection

Mobile app privacy policies serve to inform users about which kinds of personal information are collected by apps. Thus, we consider violations as *errors of omission* in that the policy failed to notify the user about a specific, collected information type.

We adopt the definition of *violation* proposed by Slavin et al. [26], which consists of: *weak violation*, which occurs when a policy refers to a vague or abstract information type that semantically includes a more specific type that was omitted from the policy; and *strong violation*, which occurs when the type is completely omitted from the policy. For example, if an app shares a user's medicine intakes, it would be considered a weak violation if the policy only states, "we collect *medical* information…" If the policy neglects to mention medical information as a collected type, then this omission is classified as a strong violation.

```
<View ... title="Set␣Check-In␣Requirements">
<View ... title="NO\_TITLE">
...
<View idName="et\_content" ... getValueOp=
"[<android.widget.EditText:android.text.Editable␣getText
    ()>]"
/>
<View ... title="Steps"/>
</View>
</View>
```

Listing 4: Partial Code of Dialog Insertion Result

When detecting strong and weak violations, for a input view v whose collected information is sent to network, we first check whether v can be mapped to a concept word c in the ontology. If so, v is considered an input view collecting sensitive information, and we further check whether c and c's ancestors in the ontology appear in the privacy policy. If neither c nor c's ancestors appear, a strong violation is detected, and if any of c's ancestors appear but c does not appear, a weak violation is detected. Slavin et al. [26]'s work uses the similar strategy, but since they focus only on information-accessing API methods, they pre-define a mapping from each API method to a concept work (e.g., mapping the method `getLastKnownLocation()` to the concept location). Such a predefined mapping is impossible for user-input data, because each app has its own set of input views and they are unknown before analysis of the app. Thus, the core technical challenge we face here is how to map an input view to a concept word in the ontology. Our approach is to develop the conceptual similarity calculation and hierarchical mapping as presented in following two subsections.

3.3.1 Concept Similarity. User-provided information types, including GUI labels, are relatively unbounded as compared to platform information types studied by Slavin et. al [26]. Thus, we consider two well-adopted similarity measurements to map GUI labels to ontology phrases: WordNet similarity [16] and Cosine similarity [25]. WordNet, which is a popular lexical database used in natural language processing, calculates similarity only for single-word pairs. To accommodate information type phrases that consist of multiple words, we propose a simple greedy alignment as follows: given phrases A and B, we align the word pairs (one in A and the other in B) that have the highest single-word similarity in WordNet, and then perform the alignment recursively until no more words in either A or B remain. For Cosine similarity, we convert the two phrase into two word vectors and apply the standard Cosine similarity formula.

Each mapping between a policy phrase and a GUI Label exists, if the similarity between the phrases, labels and the concept are higher than a given threshold, which is a parameter of our approach. We report results from evaluating this approach under different similarity thresholds in Section 4.

3.3.2 Hierarchy Mapping. 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

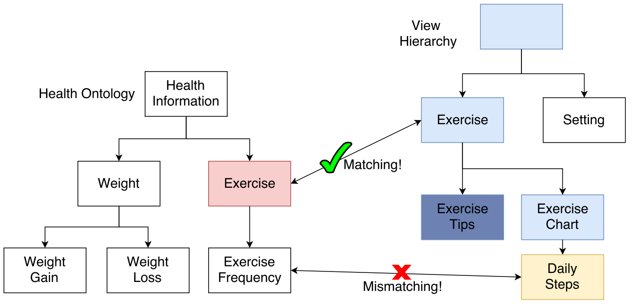


Figure 4: Illustration of Hierarchical Mapping

input view id can equate to an informative descriptions, but is often inadequate. In our motivating example, the view id of the input box is "et_content", which is too generic to be meaningfully mapped to an ontology concept. Even in three domains (health, finance and dating), and with unbounded user inputs, one still cannot exhaustively iterate all information type phrases for inclusion in the ontology.

To address this issue, we develop a novel mapping strategy that leverages the larger input context, including GUI labels, which we call *hierarchical mapping*. In contrast, we refer to the strategy of mapping only the label / id of an input view to concept words as *node mapping*. Our intuition is that, similar to ontologies, the input view hierarchy conveys information about concept relationships between GUI labels. For example, an activity with the title "Transaction Information" may contain multiple user input boxes about transaction time, source account, etc, which all are potential sub-concepts of transaction information in an ontology. Therefore, when mapping user input views to ontology phrases, we use not only the input view ID and label, but also its ancestor view IDs and labels in the view hierarchy.

As shown in Figure 4, we first collect the IDs and labels of all ancestor views for a given input view (light blue views). Next, we collect the view IDs and labels that are sibling views immediately before any collected ancestor views (the dark blue view), because sibling views may contain input view labels of their own. We refer to these collected IDs and labels, collectively, as *ancestor labels*. If an input view ID and label cannot be directly mapped to any ontology concept, we further map the ancestor labels to the ontology. Notably, the hierarchical mapping would presumably increase recall due to fewer false negatives, however, at the cost of additional false positives and lower precision. Our evaluation shows that the false positives are less significant when compared with the improvement in recall.

4 EVALUATION

We evaluate the approach using 150 apps collected from the Google Play with privacy policies in three categories: finance, health, and dating. Within finance and health, we examined apps regulated by GLBA and HIPAA, respectively, and unregulated apps for personal budgeting and personal health. In dating, we explored apps for serious dating, which often requires elaborating user profiles, and casual dating, which includes anonymous chat apps. It should be noted that, health, finance and dating are categories defined in Google Play Market. To be representative, we further classify each category to two sub-categories (personal / institution for finance, fitness / medical for health, and serious / casual for dating). To acquire the 150 apps, in the listed apps in each category at Google Play, we scan from the top until we collect 25 apps with privacy policies for each sub-category. Finance and health apps was collected in Jan 2017, and dating apps were collected in July 2017.

Table 1: Lexicon analysis

	Health	Finance	Dating	Overall
Total HITs	141	52	141	334
Average Words per HIT	105	102	116	108
Total Annotations - Crowd Workers	739	309	868	1,916
Total Annotations - Authors	456	198	508	1,162
Total Unique Information Types	197	112	262	490
Annotation Time	34.7	13.1	36	84

The highest ranked 25 apps that have privacy policies were selected for each category from Google Play using the category name as the search word. To build our three domain ontologies, we chose the five apps with longest privacy policies from each sub-category (in total 10 apps per category, 30 apps in total). The remaining 120 apps comprise the test set. All data, including the ontologies, links to apps and privacy policies, GUI XML files, anonymized survey responses and the violation detection tool can be downloaded at our anonymized project website [4].

Our empirical evaluation tries to answer the following three research questions.

- **RQ1:** What is the effort required and resulting quality from constructing an ontology?
- **RQ2:** What are the quantity and type of privacy-policy violations in apps, if any?
- **RQ3:** How do different similarity calculations and thresholds, and mapping strategies, affect violation detection?

4.1 Ontology Evaluation

In response to research question **RQ1**, we report the effort and quality of extracting the lexicon, before reporting effort and quality of constructing the ontology construction. The effort to construct an ontology consists of the time to extract the lexicons from the policies and the time to identify semantic relationships in the lexicons. Table 1 shows the total HITs to collect information type annotations, average word count per HIT, total annotations collected from crowd workers and authors, total unique information types extracted, and combined annotation time for authors and crowd workers.

Overall, the average time to extract an information type from a privacy policy in health, finance and dating is 10.6 minutes, 7.0 minutes, and 8.4 minutes, respectively. This time includes the additional time from author annotations needed to evaluate the method.

The lexicon quality is measured by the consensus between author and crowd worker annotations as measured by extracted, unique information types. In health, the authors and crowd workers agreed on 105/198 unique information types. In addition, the authors missed 55 information types that the crowd workers annotated, and the crowd workers missed 34 information types that the authors annotated. In finance, all annotators agreed on 69/112 information types, crowd workers annotated an additional 20 types, whereas authors annotated an additional 23 types. In dating domain, all annotators agreed on 135/262 information types, crowd workers annotated additional 76 information types and authors annotated 51 additional unique information types. Overall, the crowd workers generally identified 18-29% more information types, and authors generally identified 17-20% more types. The consensus was 52-62% of types extracted.

In addition to comparing annotator performance, we compared the lexicon coverage across each domain. The health and finance lexicons share 32/278 phrases, health and dating share 45/415 phrases, and finance and dating share 27/347 phrases. This is an overlap of only 8-12% across three domains, which is due to the differences in policy focus and application features.

Separate ontologies were constructed for each domain where two analysts individually identify semantic relationships between phrases in a lexicon in one domain, followed by a reconciliation step to remove conflicts between annotators. To evaluate the quality of the ontology, we used Fleiss’s Kappa to measure the degree of agreement above chance before and after each reconciliation step [9]. The average time per analyst to identify semantic relationships in health, finance and dating was 6 hours, 5 hours and 8 hours, respectively. The average time to reconcile disagreements were 3.7 hours, 2 hours and 5 hours, respectively.

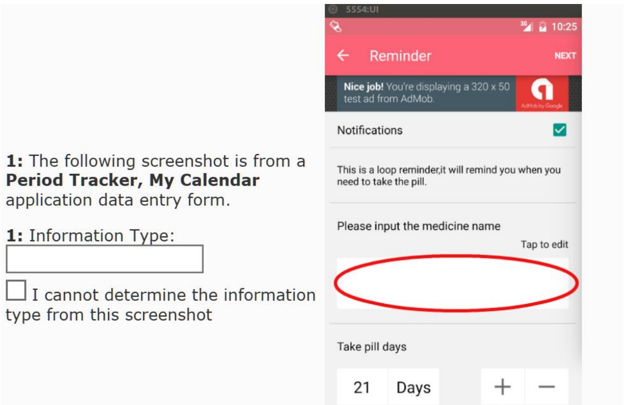
In health, the resulting $KB1$ and $KB2$ for the two analysts contain 951 and 920 axioms, respectively. We obtained these results after two rounds of comparisons and reconciliations. The first comparison produced 491 axioms in disagreement and reconciliation reduced the disagreements to 78 axioms. The Fleiss Kappa after the first and second reconciliations were 0.77 and 0.80, respectively. In finance, the resulting $KB1$ and $KB2$ for the two analysts contain 590 and 582 axioms, respectively. The first comparison produced 292 axioms in disagreement and reconciliation reduced the disagreements to 43 axioms. The Fleiss Kappa after the first and second reconciliations were 0.83 and 0.92, respectively, showing a larger increase in agreement. In dating domain, the resulting $KB1$ and $KB2$ for the two analysts contain 1,049 and 1,289 axioms, respectively. The first comparison produced 569 axioms in disagreement and reconciliation reduced the disagreements to 146 axioms. The initial Fleiss Kappa before reconciliation was 0.17 which was increased to 0.79 after the first round of reconciliation.

To evaluate the ontology construction method, two different authors, who we call examiners, independently applied the construction method to the finance lexicon. The Fleiss Kappa value comparing the analyst- and examiner-constructed ontologies was 0.54. On inspection, the disagreement is comprised of 148/474 axioms. Among the 148 axioms, 74 axioms can be inferred using a syntactic analyzer, which automates the process of ontology construction by inferring semantics from lexicon phrases based on their syntax, alone (e.g., plural-singular forms that are equivalent for our purposes). Resolving the 74 axioms with a syntactic analyzer yileds a Kappa of 0.75 between the analysts and examiners. Among the unresolved differences, the examiners found hypernymy relationships missed by the analysts, such as “deposited checks,” which are a kind of “transaction.” We believe these differences are due to (1) the various interpretations of phrases by analysts and examiners; (2) the fatigue of comparing phrases; (3) and recency effects that both analysts and examiners experienced during ontology construction [21].

4.2 Ground Truth

The research questions **RQ2** and **RQ3** depend on a *ground truth*-the correct number of true violations in the training set. Three challenges must be addressed to establish this ground truth.

First, it is impossible to know all of the true positives, and thus it is impossible to measure recall. To address this challenge, we adopt relative recall [26], which equates the set of true positives to all violations that are detectable with available techniques. In our approach, this violation set results from applying all variants of our approach to the test set, and taking the union of detected violations. Second, the true mapping from GUI labels to the ontology is comprised of the range of input field names acceptable to human interpretation, which can vary. To address this challenge, for each violation detected by any variant of our approach, we first use crowd sourcing to elicit the generally acceptable names of user input fields. Then, we followed rigorous steps to map the crowd worker interpretations of field names to the ontology concepts. Third, the information flows reported by FlowDroid need to be validated with runtime observation of information leaks to the network. We validate the information flows using the Xposed framework. In total, our ground truth construction consists of 21 strong violations and 18 weak violations from 19 apps of the test set.


1: The following screenshot is from a **Period Tracker, My Calendar** application data entry form.

1: Information Type:

□ I cannot determine the information type from this screenshot

Nice job! You're displaying a 320 x 50 test ad from AdMob.

Notifications ☑

This is a loop reminder,it will remind you when you need to take the pill.

Please input the medicine name

Tap to edit

Take pill days

21 Days

Figure 5: User Interface Input Field Tagging Task

4.2.1 Eliciting GUI Input Field Types.

We first analyzed 53 input field labels and found that only 33.9% percent correctly describe the field type. To elicit input field types from crowd workers, we designed a free listing survey [7], in which workers were asked to identify the information type that describes the information entered into the app through a specific GUI input field, shown in a red circle in the screenshot (see Figure 5). Each survey consists of 3-5 screenshots, and we surveyed 53 input fields from 19 apps. We recruited 30 participants per survey using Amazon Mechanical Turk to yield 393 HITs. Participants of the surveys were located in the United States with an overall HIT approval rating greater than 95%.

We obtained 30 information types per input field. Because there are multiple ways to describe the same concept, we pre-processed the results to more easily compare elicited types 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;” and (d) remove hyphens, e.g., “e-mail” is changed to “email.” These steps are similar to porter stemming in natural language processing, were verb conjugation is removed to make verb comparisons easier [19, 20]. After pre-processing, we combine similar type names for each field and calculate the type name frequency, which is the number of workers who provided each

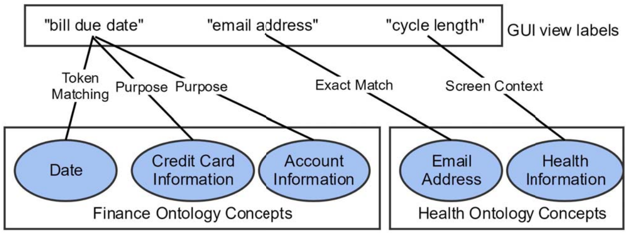


Figure 6: Mapping

syntactically unique type name per field. Finally, for each field, we select the most frequent type name, which remains linked to a set containing the less frequent type names for that field. This set can be used to expand the interpretation of information types for the same input field, and also can be used to map the GUI input field to the ontology, which is described next in Section 4.2.2.

4.2.2 Mapping. We follow a three step approach to map the true input field type names to the ontology concepts: (1) for each elicited name, we look for the exact match of the name in the ontology. If the match is found, we map the name to the matched concept in the ontology; (2) if we cannot find the exact match, (a) we break the name into separate words and create a phrase superset, which includes any combination of the individual words from the name. Next, we look for the exact match of the phrases in the superset with concepts in the ontology. If we find an exact match, we map the original name via this phrase to the matching concept in the ontology; and (b) we identify the purpose for the GUI input field name using existing concepts in the ontology, if a matching concept is found for the purpose, we match the name to that concept. (3) If we cannot find a related concept for the name using steps (1) or (2), we use the context of the screen where the input field is present in the app. The context provides guides to find related ontology concepts to the name. Figure 6 shows the mapping for elicited names from the input field in Figure 5. The phrase "bill due date" fails to find an exact match in the finance ontology in step (1), but after word tokenization in step (2) produces the word set $S = \{bill, due, date\}$. Next, the superset of S yields $T = \{bill, due, date, billdue, billdate, duedate, billduedate\}$. Finally, the generated name "date" from the superset matches "date" in the finance ontology. In step (3), the purpose for "bill due date" yields matches for "account information" and "credit card information." In a second example, we were unable to find matching concepts for "cycle length" using the two first steps. Therefore, using the context of the screen that contains the GUI input field and the application itself, we found that "cycle length" is related to "menstrual information" and not "exercise information." Therefore, we mapped "cycle length" to "health information." This process was performed by the authors who voted on the final result to construct the ground truth.

4.2.3 Validation with Xposed. We validated the FlowDroid results by implementing a runtime tool to "hijack" the apps with detected violations. To do so, we created a module that utilized the Xposed framework, which is depicted in Figure 7. The Xposed framework can modify compiled Android apps 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 is able to insert hooks into the bytecode of the app allowing the module to perform custom code before and after hooked method calls. 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. Our module inserts custom code before invocations of network sink API methods, and the code 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 GUI input values were leaked by reviewing log files.

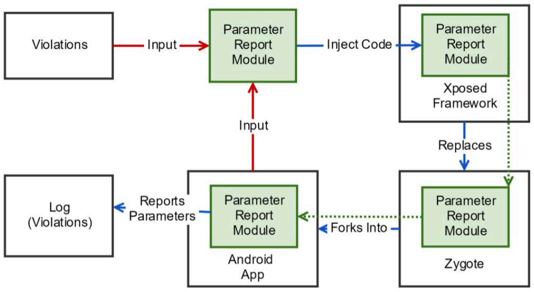


Figure 7: Xposed Parameter Reporter Module

4.3 Violation Detection Results

We designed several variants of our approach based on different concept similarity calculations, similarity thresholds, and mapping strategies, which we evaluated using the ground truth. We use *precision*, *relative recall* and *F-score* as our metrics. In our experiment, we consider two similarity measurements: WordNet (WN) and Cosine similarity (Cos). We consider two mapping strategies: node-mapping (Node), wherein only the ID and label of the input view is considered, and hierarchical-mapping (Hier), where IDs and labels of all ancestor input views are considered. This yields four approach variants by combing techniques: Hier+WN, Hier+Cos, Node+WN, Node+Cos.

The violation detection results are presented in Figure 8. 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 compare the precision, relative recall, and F-score on strong violations respectively. The figures in row 2 compare the precision, relative recall, and F-score, respectively, on violations with strict violation types (e.g., strong violations are detected as strong, and weak violations are detected as weak). The figures in row 3 compare the precision, recall, and F-score on violations with general types, respectively. By general types, we mean a violation is considered correctly detected if a true violation is detected but the predicted type is wrong (e.g., strong violations detected as weak, and weak violations detected as strong).

From Figure 8, we have the following observations. First, with a 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, where violation-type errors are ignored so better results are achieved.

Second, hierarchical-mapping variants (solid lines) perform much better (on average, improved by 20 percent in recall, and 13 percent in F-score) than node-mapping variants in both recall and F-score,

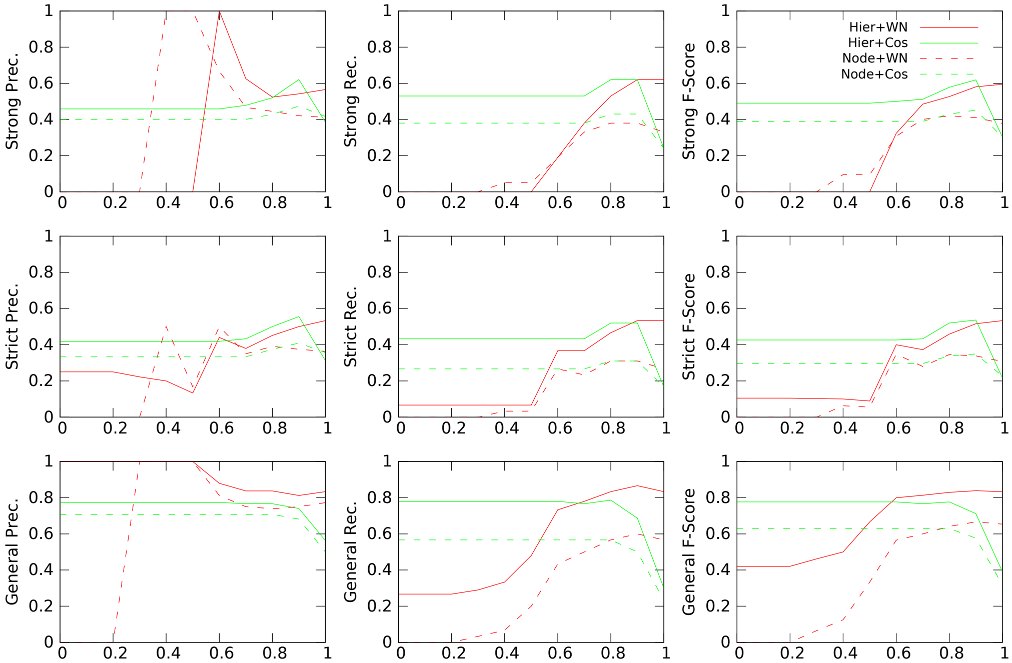


Figure 8: Comparison of Technique Variants under Different Similarity Thresholds

while the precision of the two techniques are similar. This observation confirms our intuition that the incorporation of context GUI labels greatly improve the violation detection results (note that recall is more important than precision in this scenario, as long as the precision difference is small).

Third, as the similarity threshold increases, the effectiveness of WordNet-based variants improves while the effectiveness of Cosine-based variants reduces. With the similarity threshold around 0.8, all variants are close to their best performance (F-score). One explanation for this observation is that WordNet often provides high similarity scores to words that are not closely related in the domain, but which may be closely related in other domains such as bank and river. Thus, increasing the similarity threshold reduces these false positives. By contrast, Cosine similarity requires matching exact words, and a high similarity threshold will result in losing matches between GUI labels and ontology concept names.

Examples. We hereby describe some real examples about privacy violations. To avoid legal issues, we do not reveal the name of 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 sends information about cycles and medicines taken to their servers, but it does not mention sharing this information in the privacy policy. Other strong violations include the unmentioned collection of food and weight information from two diet apps, and the unmentioned collection of insurance information from a pregnancy tracking app, etc. By contrast, a weak violation was found in a top personal budgeting app with more than 1 million installs. We found that this app sends the bill due date information to the server, but it is not directly mentioned in the privacy policy, although "transactions" as a more general concept phrase is mentioned in the policy. Other interesting weak violations include steps goals and steps taken for exercise in a fitness app (only exercise information is mentioned), etc.

4.4 Threats to Validity

On construct validity, the claim that a mobile app violates a privacy policy requires a semantic mapping between the code and a corresponding policy statement. The mapping consists of information type phrases in policies, labels of input fields in mobile app input views, information flows extracted from source code, and formal ontology concepts that align these artifacts. To address this threat, we crowd sourced the identification of relevant policy phrases and for information types of input fields as seen by potential users. In the construction of the lexicon and ontology, we computed inter-rater reliability and compared results across crowd workers and two sets of authors, called analysts and examiners. The method results show a high, above-chance agreement and the method reveals specific sources of disagreement.

On internal validity, the lexicons were constructed from two or more annotators, which yield information types that are acceptable to a subset of annotators, but not all annotators. The liberal interpretation may have skewed our results to include more false positives, in which a privacy policy phrase has a statistically narrower interpretation accepted by the general population than what was accepted in the ontology. In addition, the existing frameworks (e.g., FlowDroid and GATOR) also have deficiencies in precision and recall, which are inherited by our framework. For example, FlowDroid's performance on DroidBench, a benchmark repository of mobile apps, yields 93% recall and 86% precision on data leak detection [6]. To address this threat, we carefully reviewed the flows reported by these frameworks when evaluating the precision and recall of our overall approach.

External validity concerns how well our framework generalizes to other mobile apps and domains. In our study, we chose three different domains aimed at highlighting cross-domain differences: health, finance and dating. In addition to different information types, apps in this domains also present different kinds of input forms and views that were designed to meet different domain-specific features (e.g., diet and exercise versus bank account balances versus social networks).

5 DISCUSSION

Bytecode Analysis and Monitoring. Security and privacy research aims to address a specific threat, which in our case is the carelessness of software developers to recognize privacy policy violations in code. Therefore, we assume that these developers have access to the source code, to which they can apply source code analysis and instrumentation techniques. However, in our approach, we choose to perform bytecode analysis and platform-based monitoring, based on the following three reasons. First, our approach allows direct analysis and monitoring of unmodified APK files, which extends access to our framework to a broader community as a third-party service. Thus, developers, project managers, or even regulators can use the framework without access to the source code to check for policy violations. Second, platform-based monitoring is independent from app code changes, which is easier for developers who would not need to re-instrument their code after changes to the app. Third, the existing tools Soot and Xposed reduce the technical difficulty of performing bytecode analysis and platform-based monitoring in comparison to source code analysis and instrumentation.

Limitations on data types. Our approach employs FlowDroid for information flow analysis and SUSI for network sinks. These tools have limitations when detecting flows and network requests involving encrypted data and files. For example, our approach currently cannot detect when the user-provided data is stored in a file and sent out through a network request. With respect to encryption, we consider HTTPS API methods as sinks and, if encryption is performed through HTTPS, our approach can detect the violation. However, if the data is encrypted within the app, FlowDroid may not be able to track the data through data encrypting methods. We believe these limitations can be addressed by improvements so information flow analysis.

6 RELATED WORK

To our knowledge, GUILeak is the first approach that uses data flow analysis to verify consistency between app-collected data and privacy policy text with regard to application code and user input. The following are related works in the area of Android data flow analysis and privacy policies.

Slavin et al. [26] and Zimmeck et al. [33] used similar approaches to detect privacy policy violations in Android apps based on Android API calls. Such an approach is useful in identifying leaks where the API calls collect personal information from a mobile device. Yu et al. [30] developed an approach to detect policy violations with more advance data flow models. There are also works on describing information manipulation behavior of applications. DESCRIBEME, developed by Zhang et al. [32], is another tool that automatically generates security-centric description and bridges the gap between permissions and descriptions. It helps end users to better understand what information types have been collected through permissions. Yu et al. [31] developed a novel approach to generate privacy policy text from application code and permission profile. Different from these approaches which focus on privacy information collected from API methods, GUILeak can detect violations on information collected from GUI. By mapping privacy-policy phrases to user input views, we are able to go beyond Android API-based violation 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 efforts on GUI-related information collection explore various facets of privacy and security. AsDroid [15] match text from GUI components to top-level functions in order to detect clandestine behavior. Their work targets functions by identifying suspicious permissions. In contrast, our work compared the consistency between privacy policies and user input via GUI components which are based on native code. This allows our approach to not be limited by the coarse granularity of Android permissions. SUPOR [13] and UIPicker [18] identify sensitive input views to which sensitive information can be entered. Compared with these efforts, (1) besides detecting unusual information collections, GUILeak further checks whether the information collection is mentioned in the privacy policy based on ontology and mapping techniques to map GUI views to privacy terms, and (2) on code analysis, GUILeak adapts Gator to extract contexts of input fields in dynamically generated dialogs and layouts as shown in Section 2. In contrast, SUPOR and UIPicker extract context from static layout xml files.

BIDTEXT [14] is a tool for reporting the propagation of label set variables corresponding to sensitive text labels to sinks. It examined text labels from either code or GUI and relied on a keyword set to determine the sensitiveness of computed texts. But BIDDTEXT is not able to solve our problem because it does not try to map sensitive labels to phrases in the privacy policies. On input label analysis, generalized taint analysis [27, 28] and collaborative hybrid analysis [29] are existing approaches to locate user-visible constant strings. Compared with these approaches, our GUI analysis component needs to further differentiate labels corresponding to input views.

7 CONCLUSION

In this paper, we proposed a novel approach to detect privacy policy violations due to leak of user input data. To address 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 three important domains (finance, health and dating) and detected 21 strong violations and 18 weak violations in 120 popular apps from the domains. Our experiment shows that our best technique variant can achieve a F score of 84% on general violation detection with proper similarity threshold set.

ACKNOWLEDGMENT. The authors are supported in part by NSF Awards CNS-1330596, CCF-1464425, CNS-1748109, NSA Grant on Science of Security, and DHS grant DHS-14-ST-062-001.

REFERENCES

[1] Google Play Statistics, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed: 2017-08-23.

[2] International Data Corporation (IDC) Smartphone OS Market Share 2017 Q1, http://www.idc.com/promo/smartphone-market-share/os. Accessed: 2017-08-23.

[3] Mint by the Numbers: Which User Are You?, https://blog.mint.com/credit/mint-by-the-numbers-which-user-are-you-040616/. Accessed: 2017-08-23.

[4] UI Privacy Project Web Site, https://sites.google.com/site/uiprivacy2017/. Accessed: 2017-02-22.

[5] Xposed Framework, http://repo.xposed.com. Accessed: 2017-02-22.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (Jun 2014), 259–269. DOI:http://dx.doi.org/10.1145/2666356.2594299

[7] Harvey Russell Bernard. 2011. *Research methods in anthropology: Qualitative and quantitative approaches*. Rowman Altamira.

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

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

[10] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Proc. 10th International Static Analysis Symposium (SAS) (LNCS)*, Vol. 2694. Springer-Verlag, 1–18. Available from http://www.brics.dk/JSA/.

[11] Senate Banking Committee. 1999. Gramm-Leach-Bliley Act. (1999). Public Law 106-102.

[12] Mitra Bokaei Hosseini, Sudarshan Wadkar, Travis D Breaux, and Jianwei Niu. 2016. Lexical Similarity of Information Type Hypernyms, Meronyms and Synonyms in Privacy Policies. In *2016 AAAI Fall Symposium Series*.

[13] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 977–992. http://dl.acm.org/citation.cfm?id=2831143.2831205

[14] Jianjun Huang, Xiangyu Zhang, and Lin Tan. 2016. 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)*. ACM, New York, NY, USA, 169–180. DOI:http://dx.doi.org/10.1145/2950290.2950348

[15] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. 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)*. ACM, New York, NY, USA, 1036–1046. DOI:http://dx.doi.org/10.1145/2568225.2568301

[16] Adam Kilgarriff and Christiane Fellbaum. 2000. WordNet: An Electronic Lexical Database. (2000).

[17] Paul Krebs and T. Dustin Duncan. 2015. Health App Use Among US Mobile Phone Owners: A National Survey. *JMIR mHealth uHealth* 3, 4 (04 Nov 2015), e101. DOI:http://dx.doi.org/10.2196/mhealth.4924

[18] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. UIPicker: User-input Privacy Identification in Mobile Applications. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 993–1008. http://dl.acm.org/citation.cfm?id=2831143.2831206

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

[20] Martin F Porter. 2001. Snowball: A language for stemming algorithms. (2001).

[21] Leo Postman and Laura W Phillips. 1965. Short-term temporal changes in free recall. *Quarterly journal of experimental psychology* 17, 2 (1965), 132–138.

[22] Siegfried Rasthofer, Steven Arzt, Ec Spride, Technische Universitt Darmstadt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. (Feb 2014).

[23] Health Resources and Services Administration. 1996. Health Insurance Portability and Accountability Act. (1996). Public Law 104-191.

[24] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 143, 11 pages. DOI:http://dx.doi.org/10.1145/2544137.2544159

[25] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[26] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. 2016. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 25–36. DOI:http://dx.doi.org/10.1145/2884781.2884855

[27] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2009. Locating need-to-translate constant strings for software internationalization. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 353–363.

[28] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2009. Transtrl: An automatic need-to-translate string locator for software internationalization. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 555–558.

[29] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 16.

[30] Le Yu, Xiapu Luo, Chenxiong Qian, Shuai Wang, and Hareton KN Leung. 2017. Enhancing the description-to-behavior fidelity in android apps with privacy policy. *IEEE Transactions on Software Engineering* (2017).

[31] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. 2017. Toward Automatically Generating Privacy Policy for Android Apps. *IEEE Transactions on Information Forensics and Security* 12, 4 (2017), 865–880.

[32] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. 2015. Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 518–529. DOI:http://dx.doi.org/10.1145/2810103.2813669

[33] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M. Bellovin, and Joel Reidenberg. 2017. Automated Analysis of Privacy Requirements for Mobile Apps. In *Network and Distributed System Security Symposium NDSS*.