Toward a Framework for Detecting Privacy Policy Violation in Android Application Code

Rocky Slavin\textsuperscript{1} rocky.slavin@utsa.edu Xiaoyin Wang\textsuperscript{1} xiaoyin.wang@utsa.edu Mitra Bokaei Hosseini\textsuperscript{1} mitra.bokaeihosseini@utsa.edu

William Hester\textsuperscript{2} william.hester@utdallas.edu Ram Krishnan\textsuperscript{3} ram.krishnan@utsa.edu Jaspreet Bhatia\textsuperscript{1} jbhatia@cs.cmu.edu

Travis D. Breaux\textsuperscript{4} breaux@cs.cmu.edu Jianwei Niu\textsuperscript{1} jianwei.niu@utsa.edu

\textsuperscript{1}Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas, USA

\textsuperscript{2}Department of Computer Science
University of Texas at Dallas
Richardson, Texas, USA

\textsuperscript{3}Department of Electrical and Computer Engineering
University of Texas at San Antonio
San Antonio, Texas, USA

\textsuperscript{4}Institute for Software Research
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

ABSTRACT

Mobile applications frequently access sensitive personal information to meet user or business requirements. Because this information is sensitive, regulators increasingly require mobile app developers to publish privacy policies that describe what information is collected, for what purpose is the information used and with whom it is shared. Furthermore, regulators have fined companies when these policies are inconsistent with the actual data practices of mobile apps. To help app developers check their privacy policies against their apps for consistency, we propose a semi-automated framework that consists of a policy terminology-API map that links policy phrases to API functions that process sensitive information, and information flow analysis to detect misalignments. We present our results from a collection of API to policy phrase mappings followed by a case study of 501 top Android apps that discovered 63 potential privacy policy violations.

1. INTRODUCTION

In early 2015, the Android operating system (Android) account for 78.0\% of the worldwide smartphone market share [5]. With this increase in market share comes an increase to end user privacy risk as mobile applications (apps) built for the Android have access to sensitive personal information about users’ locations, contacts, and unique device information. To protect privacy, regulators, such as the U.S. Federal Trade Commission (FTC), have relied on natural language privacy policies to enumerate how applications collect, use, and share personal information. Recently, the California Attorney General Kamela Harris negotiated with the Google Play app store to require mobile app developers to post privacy policies [19]. Despite this effort to produce these policies, as with any software documentation, there are opportunities for these policies to become inconsistent with the code. These policies can be written by people other than the developers, such as lawyers, or the code can change while the policy remains static. Such inconsistencies regarding an end user’s personal data, intentional or not, can have legal repercussions that can be avoided with proper consistency checks. For example, the FTC, under their unfair and deceptive trade practices authority, requires companies to be honest about their data practices their privacy policies. Companies, such as SnapChat, Fan- dango, and Credit Karma, often settle with the FTC for inconsistent policies and practices by accepting 20 years of costly privacy and security audits [1, 2]. It is therefore good practice for mobile apps to clearly state in their privacy policies what data is collected and for what purpose. For large companies, this task is commonly assigned to a team of legal experts, however, mobile app developers are frequently small start ups with 1-5 developers [17]. It is important for software engineers to be aware of the data their code is collecting along with what their policy says they are collecting not only for legal reasons, but for the production of quality apps. As more data is entrusted to technology, end users are becoming more aware of the ramifications of mishandled private data [25]. Thus, software engineers are entrusted by the end user to not only care for their data, but disclose what exactly is being collected.

In this paper, we present three contributions: (1) an empirically constructed mapping from policy phrases to private-data-producing Android Application Program Interface (API) functions that has been compiled from real-world apps, policies, and API documentation. The weighted many-to-many map links 38 commonly used data collection phrases and their synonyms to 267 Android API function names. (2) We created an approach that identifies privacy promises in mobile app privacy policies and checks these against code using information flow analysis to raise potential policy violations. As part of checking for data over-collection violations within the app, the approach uses information flow analysis to see if the data is sent outside the app. (3) We constructed an initial ontology of data collection phrases to use in conjunction with the API
mappings. The ontology provides a means to increase the phrase coverage of the mappings without the need for analysis on more apps and privacy policies.

This paper is organized as follows: in Section 2, we review the background upon which we based our approach; in Section 3, we describe the approach we used for our framework; Section 4 describes the evaluation of our approach followed by discussion of the results and approach in Section 5; Section 6 includes related work; and we conclude and discuss future work in Section 7.

2. BACKGROUND

This section presents the background upon which our research is based.

2.1 Android Operating System

Android is an open source mobile operating system (OS) based on more than 100 open source projects like Linux kernel. Android is developed by Google and has been reported more popular as target platform for developers than iOS in 2015 which makes it the most popular mobile operating system today [3]. Apps made to run on Android can be downloaded from multiple repositories, the most popular being Google Play 1. In 2013, Google revealed that there were more than 538 million Android users and over one billion active monthly users. These characteristics make it attractive to startups and established companies alike.

Android utilizes the Linux security model and layers through a user-based permission system [35]. Apps can access resources through the permission system to gain access to resources such as the camera, GPS, Bluetooth, telephony functions, network connections, and other sensors [35]. Such permissions are granted to apps by users when they install the app. All permissions not listed to and subsequently granted by the user are denied to the app [34]. Although Android applies this permission system and rigorous security management, data leakage and misuse is still possible [13]. This can be due to problems with the current Android permission system such as low granularity of the permissions [24] and the ambiguity of the phrases presented to users when installing an app [22]. Problems such as these can allow apps to access the sensitive data by calling Android Application Program Interface (API) in the app source code.

2.2 Application Program Interface

Android 4.2 contains over 110,000 public APIs and some of them are specifically used to retrieve, insert, update, or delete sensor data through the Android OS [27]. APIs increase the level of security by not allowing apps to have direct access to all sensor data by default.

Before an app can access APIs, the required permissions must be requested by the app through a manifest file. An app’s manifest file enumerates the app’s required permissions and is described to users when installing an app as well as on the app’s download page on the Google Play store. Thus, there is a direct relationship between the permissions granted to an application by a user at installation time and eligible API calls in the application source code.

2.3 Privacy Policy

Besides the standard permissions for API calls documented in manifest files, applications’ privacy policies are a source for identifying what information is collected and used by apps. A privacy policy serves as the primary means to communicate with users regarding which and how sensitive personal information (SPI) has been accessed, collected, stored, shared (app to app, and to third party) and used/processed, and the purpose of the SPI collection and processing. Privacy policies generally consist of multiple paragraphs of natural language such as the following excerpt from the Indeed Job Search app’s privacy policy 2 listed on Google Play:

Indeed may create and assign to your device an identifier that is similar to an account number. We may collect the name you have associated with your device, device type, telephone number, country, and any other information you choose to provide, such as user name, geo-location or e-mail address. We may also access your contacts to enable you to invite friends to join you in the Website.

The reason that privacy policies are so important is that the United States takes a “notice and choice” approach to address privacy online [28]. Under this framework, app companies post their privacy policies; users read the policies and make informed decision on accepting the privacy terms before installing the apps [28]. However, most privacy policies prepared by policy authors are difficult to understand due to their verbose and ambiguous nature, and this can lead to users to skip reading policies even if they have concerns about information collection practices. More significantly, the app developers might not be able to comply with privacy policies effectively. To address this issue, this work aims to provide a framework to achieve alignment between apps’ privacy policies and implementation code, and better communication among software developers and policy writers.

A major hindrance in the understanding and analysis of privacy policies is that there is no canonical format for presenting the information. The language, organization, and detail of policies can vary from app to app.

3. APPROACH

The goal of this work is to discover information regarding the relationship between terminology used in privacy policies expressed in natural language and API calls used in the corresponding code. Such a mapping would then provide semantic information regarding the natural language. In turn, an app’s source code could more easily be checked for misalignment with its corresponding privacy policy. Further, more precise policies could be generated from source code based on feedback from our framework. The following subsections describe our use of data retrieval, information flow analysis, and ontology construction to construct a framework for bridging the gap between natural language privacy policies and their corresponding implementations.

3.1 API Mapping

Figure 2 presents an overview of the approach that we developed to construct the mapping of data collection phrases to Android APIs. We start from the apps’ policies and bytecode to generate an empirical-based mapping, stored in the Mappings src database. We then utilize the Google API documentation to create the mapping database, Mappings doc, which relates the phrases appearing in the apps’ privacy policies to the APIs containing those phrases in the document. Both mappings takes the data collection lexicon as an input. We first describe the process used in the lexicon creation shown in Figure 1 in the following subsection.

1https://play.google.com/

2http://www.indeed.com/legal
3.1.1 Data Collection Phrases Lexicon Construction

Each app page on Google Play includes a link to the app’s privacy policy if it is specified by the developer. We created a Python script to download the privacy policies from these links for as many of the top 1000 apps on the Play Store. We were able to collect 501 privacy policies and selected 50 of them based on their formatting, language (we only considered policies written in English), and whether or not a “Privacy Policy” section was explicitly stated in the document (see Table 1).

Prior to running the task, we prepared the policies for input to our phrase collection process. First, we itemized the policies into paragraphs. During the itemization process, we remove sections describing the following content: the introduction and table of contents, “contact us,” security, U.S. safe harbor, policy changes, and California citizen rights. This content generally appears in separate sections or paragraphs, which reduces the chance of inconsistency when removing these sections across multiple policies. Finally, we manually split the remaining policy into spans of approximately 120 words. We preserve larger spans which either has an anaphoric reference back to a previous sentence (e.g. when “This information,...” depends on a previous statement to understand the context of information), or when the statement has subparts (e.g., (a), (b) etc.) that depend on the context provided by earlier sentence fragments. On average, we need 15 minutes to divide each policy.

Next, we conducted an online annotation task to identify information types in 50 policies. The task required each of the five experts and knowledgable users to read a short, approximately 120 word, paragraph from a privacy policy and to identify relevant phrases. Specifically, they were asked to identify: platform information - any information that the mobile app provider or another party accesses through the mobile device and platform API, which is not unique to the app; or other information - any information that the mobile app provider or another party collects.

Every participating expert reviewed each of the randomized policy fragments independently. For each fragment, the participant annotated phrases related to data collection regarding specific information that would be collected. The resulting phrases were compiled and only those that were chosen by at least two participants were kept. Table 2 lists the initial set of phrases collected. Then the five participating experts (from our research team) were called to the meetings to sanitize the resulting phrases based on following criteria. First, we filtered out phrases that were not considered API-related. For example, the phrase “chat text” was removed because it was something that would not generally be retrieved by an Android API (i.e., it is something that an app’s native code would do). Second, we filtered phrases that were not platform related. This kind of information is common to many different devices. For example, the phrase “contact” is not related to the device’s platform but would remain on the phrase list after the previous filtering. Table 3 lists the remaining phrases after these filters. We also handled equivalent phrases by creating a separate list of synonyms, initialisms, and acronyms to keep track of phrases that meant the same thing. Plural forms of words were also marked as equivalent to their singular forms. The final lexicon consists of 38 commonly used data collection phrases and their equivalent phrases.

3.1.2 Empirically-Based Mapping

We used the lexicon as a filter through which we extracted data collection phrases from privacy policies and associated them with their corresponding real-world applications. This subsection describes the process for generating these empirically-based mappings.

We limited the APIs we analyzed to those which fell into spe-
### Table 1: 50 policies

http://mymixtapez.me/privacy-policy.html  
https://www.iubenda.com/privacy-policy/494815  
http://www.playtika.com/privacy-policy.html  
http://ubimobile.com/termsanduse/index.html  
http://www.adobe.com/special/misc/privacy.html  
http://www.apusapps.com/launcher/privacypolicy.html  
http://www.blizzard.com/company/privacy.html  
http://cardinalblue.com/privacy  
http://www.cmcm.com/about/privacy.html  
http://www.cmcm.com/protocol/cmsecurity/privacy.html  
http://terms.withhive.com/terms/mobile/policy.html  
http://www.creditkarma.com/about/privacy  
http://www.crowdstar.com/privacy  
http://en.papayamobile.com/p/privacy-policy  
http://www.fingersoft.net/privacy.html  
http://privacy.ea.com/en  
http://loveemojikeyboard.blogspot.jp/2014/10/emoji-keyboard-privacy-policy-1.html  
http://www.fungames-forfree.com/privacy  
http://www.google.com/mobile/privacy.html  
http://www.google.com/wallet/privacy.html  
https://ringtone-maker.appspot.com/FAQ.html  
http://iigg.com/about/privacy_policy.php  
http://www.imangistudios.com/privacy.html  
https://imo.im/privacy  
http://www.indielegal.com/legal/privacy/  
http://en.papayamobile.com/p/privacy-policy  
http://www.firebase.com/privacy  
http://www.gameofwarapp.com/privacypolicy.html  
https://signup.netflix.com/PrivacyPolicy  
http://www.pinger.com/privacy  
http://www.vertx.com/privacy  
http://www.vertx.com/company/privacy.html  
http://www.emoji-keyboard.com/xn/PrivacyPolicy.html

### Table 2: Initial Collection Phrases

<table>
<thead>
<tr>
<th>Non-personal data</th>
<th>Personal data</th>
</tr>
</thead>
<tbody>
<tr>
<td>anonymized information</td>
<td>web browser software information</td>
</tr>
<tr>
<td>ip address</td>
<td>clicks</td>
</tr>
<tr>
<td>location</td>
<td>geo-location</td>
</tr>
<tr>
<td>operating system type</td>
<td>mac address</td>
</tr>
<tr>
<td>anonymous user usage data</td>
<td>operating system</td>
</tr>
<tr>
<td>browser type</td>
<td>page views</td>
</tr>
<tr>
<td>contacts</td>
<td>posted reviews</td>
</tr>
<tr>
<td>country</td>
<td>searches</td>
</tr>
<tr>
<td>device type</td>
<td>taps</td>
</tr>
<tr>
<td>internet protocol</td>
<td>analytics information</td>
</tr>
<tr>
<td>ip addresses</td>
<td>device</td>
</tr>
<tr>
<td>name</td>
<td>device identifier</td>
</tr>
<tr>
<td>technological information</td>
<td>location data</td>
</tr>
<tr>
<td>telephone number</td>
<td>log file information</td>
</tr>
<tr>
<td>web request</td>
<td>web pages</td>
</tr>
<tr>
<td>crash reports</td>
<td>access times</td>
</tr>
<tr>
<td>dump reports</td>
<td>browser information</td>
</tr>
<tr>
<td>location information</td>
<td>log information</td>
</tr>
<tr>
<td>mcc</td>
<td>scores</td>
</tr>
<tr>
<td>statistics</td>
<td>access devices</td>
</tr>
<tr>
<td>usage</td>
<td>browser types</td>
</tr>
<tr>
<td>usages statistics</td>
<td>chat text</td>
</tr>
<tr>
<td>version</td>
<td>device’s mac address</td>
</tr>
<tr>
<td>device identifiers</td>
<td>domain names</td>
</tr>
<tr>
<td>log files</td>
<td>landing pages</td>
</tr>
<tr>
<td>sms messages</td>
<td>physical location</td>
</tr>
<tr>
<td>advertising identifier</td>
<td>platform type</td>
</tr>
<tr>
<td>geo-location information</td>
<td>possibly ip address</td>
</tr>
<tr>
<td>browser language</td>
<td>similar information</td>
</tr>
<tr>
<td>crashes</td>
<td>technical information</td>
</tr>
<tr>
<td>device-specific information</td>
<td>urls</td>
</tr>
<tr>
<td>event information</td>
<td>voice communications</td>
</tr>
<tr>
<td>gps</td>
<td>general geographic location</td>
</tr>
<tr>
<td>hardware settings</td>
<td>phone number</td>
</tr>
<tr>
<td>information</td>
<td>device’s internet protocol</td>
</tr>
<tr>
<td>search queries</td>
<td>real-time location-based information</td>
</tr>
<tr>
<td>sensors</td>
<td>bandwidth</td>
</tr>
<tr>
<td>system activity</td>
<td>carrier network</td>
</tr>
<tr>
<td>applications</td>
<td>device capability</td>
</tr>
<tr>
<td>coarse location</td>
<td>device model number</td>
</tr>
<tr>
<td>data</td>
<td>manufacturer’s name</td>
</tr>
<tr>
<td>device id</td>
<td>platform</td>
</tr>
<tr>
<td>hardware</td>
<td>file requested</td>
</tr>
<tr>
<td>in-game information</td>
<td>internet service provider</td>
</tr>
<tr>
<td>mac</td>
<td>unique identifier</td>
</tr>
<tr>
<td>non-personal data</td>
<td>unique identifiers</td>
</tr>
<tr>
<td>software</td>
<td>domain servers</td>
</tr>
<tr>
<td>aggregated data</td>
<td>operating system information</td>
</tr>
<tr>
<td>aggregated user data</td>
<td>telecommunications carrier information</td>
</tr>
<tr>
<td>cookies</td>
<td>unique device identifiers</td>
</tr>
<tr>
<td>demographic patterns</td>
<td>geographic data</td>
</tr>
<tr>
<td>device’s uid</td>
<td>high-level geographic information</td>
</tr>
<tr>
<td>devices</td>
<td>spotify application version</td>
</tr>
<tr>
<td>mobile device</td>
<td>technical data</td>
</tr>
<tr>
<td>mobile device identifiers</td>
<td>third party applications</td>
</tr>
<tr>
<td>session data</td>
<td>transactional information enabling</td>
</tr>
<tr>
<td>tablet</td>
<td>digital rights management</td>
</tr>
<tr>
<td>mobile device model</td>
<td>internet protocol address</td>
</tr>
<tr>
<td>unique browser</td>
<td>unique device id</td>
</tr>
</tbody>
</table>
Specific categories of over 110,000 Android API sources (i.e., methods that produce data) and sinks (i.e., methods that receive data) that were categorized by the machine-learning tool SUSI [27]. Specifically, we focused on the following source categories: LOCATION_INFORMATION, NETWORK_INFORMATION, and UNIQUE_IDENTIFIER. We selected these categories because all of the phrases in the lexicon could be categorized into at least one of them.

The upper portion of Figure 2 shows the process from an input of the top free apps from the Google Play Store to a set of phrase to API mappings. For each app, if a phrase appeared in its privacy policy that was also present in our lexicon of collection phrases, we extracted it along with all of the APIs in the source code that also appeared in the SUSI categories we isolated. This resulted in an initial mapping of collection phrases to APIs for each individual app. We then generated a database of mappings for all phrases and their associated APIs, $Mappings_{src}$.

As a first step to extracting the most relevant API to phrase mappings, we analyzed the frequency at which APIs occurred across all of the apps that contained a given phrase. For example, 245 apps were found to include the word “location” (or words synonymous thereto) in their corresponding privacy policy. We then compiled a list of all APIs that were called by these 245 apps and were also included in the SUSI selected source categories along with the cumulative number of times each API was called. Comparing this number with the total number of API calls among the related apps provides a Term Frequency (TF) for the particular API (see equation 1). For example, the android.location.Location.getLongitude() API was called 1231 times in total among the 245 apps related to the word “location”. In total, there were 23,233 API calls among these apps. This results in a TF of 0.053846 for the android.location.Location.getLongitude() API with regard to the word “location”. This initial set of data gave a rough representation of correlation between phrases and APIs.

$$TF(t) = \frac{\text{total occurrences of term } t \text{ in the document}}{\text{total terms in a document}}$$

The frequency of APIs alone is not enough to provide accurate relationships since it is possible for certain APIs to simply be very commonly used in all apps. A TF score alone can be easily skewed by the API’s general popularity. To account for this, we calculated the Inverse Document Frequency (IDF) of each API (see equation 2). The IDF represents a weight that can be applied to an API’s TF in order to normalize the API’s correlation to a phrase based on how commonly it is used across all source codes. The resulting TF*IDF scores of API’s can be compared to more accurately show the correlation of an API to a phrase.

$$IDF(t) = 1 + \log_e \left( \frac{\text{total documents}}{\text{total documents containing term } t} \right)$$

As shown in blue in Figure 2, $Mappings_{src}$ was the final set of mappings of all of the isolated SUSI category APIs that appeared in the source code of all of the apps we analyzed corresponding to all phrases that appeared in apps’ respective privacy policies ranked by TF*IDF.

$$IDF(t) = 1 + \log_e \left( \frac{\text{total documents}}{\text{total documents containing term } t} \right)$$

### 3.1.3 Documentation-Based Mapping

Figure 2 shows that $Mappings_{src}$ could easily contained some mappings that were obviously not relevant (e.g., “ip address” $\rightarrow$ getLongitude). Generally, these mappings ranked low. However, we found that, while better than TF alone, TF*IDF correlations were not strong enough since many common APIs still ranked highly in terms of TF*IDF due to their disproportionately high TF.
scores. In these cases, even high IDF values for APIs with lower TFS were not enough to displace the commonly-called APIs. For this matter, we sought to include a separate approach for removing irrelevant APIs from the rankings.

Google includes publicly accessible documentation for all of Android’s public APIs with the Android software development kit. This includes descriptions of classes, methods, and fields in natural language for the use of developers. The documentation provides a direct relationship from the natural language privacy policies to the natural language documents. A documentation-based mapping would provide a relationship based strictly on natural language. However, we assert that such a phrase-based mapping would not alone fulfill the goal of our research due to its lack of empirical data. For example, the phrase “location” may be highly relevant to an API such as getAltitude, but in practice, getAltitude may rarely be used for data collection. However, the documentation contains enough information to be applied to our existing to decrease the added noise of highly-common, yet irrelevant, phrases.

In order to leverage Android’s web-based documentation, we employed the use of Google’s search API over the Android documentation. We implemented a Java-based web crawler to search for each collection phrase in our lexicon and collect the first ten results from the Android Documentation database. The basic flow is described in the lower half of Figure 2. The top results were documentation for the classes composing the relevant APIs. In some cases, results regarding packages and development articles were removed. For each of the remaining classes, we mapped their APIs to the phrases used in the search. We composed the overall results as Mappings doc.

3.1.4 Data Intersection

By themselves, the empirically-based and documentation-based mappings contain noise that makes their raw results less useful for understanding what APIs particular phrases and terms tend to represent. Figure 3 shows how the mappings resulting from the outputs of Figure 2 can be intersected (see Equation 3) to result in a more accurate representation of the APIs mappings in practice.

\[
\text{Mappings}_{\text{final}} = \text{Mappings}_{\text{src}} \cap \text{Mappings}_{\text{doc}} \tag{3}
\]

For the phrase “location”, the blue portion of Figure 2 shows that the API getIp() occurred at least once in an app with that phrase. However, intuition and a low TF*IDF value dictate that the API is likely not relevant. For the same phrase, the example in the red portion of if Figure 2 shows relevant APIs based on natural language descriptions (i.e., APIs that have semantic, though not necessarily empirical, relevance to a phrase). Figure 3 shows that the intersection removes such semantically irrelevant APIs to produce Mappings_{final}. On the other hand, starting from Mappings_{doc} in the blue portion of Figure 2, it can be seen how an API may be relevant to a phrase based on a description of the API though not relevant to the phrase in practice with regard to privacy policies. For example, as mentioned in Section 3.1.3, the API getAltitude is an API that is relevant to the phrase “location” but is not shown to be used commonly when an app’s privacy policy mentions the phrase. We remove these instances in order to create a more meaningful set of mappings based on common practices. Filtering Mappings_{doc} through Mappings_{src} results in mappings that are semantically and empirically relevant. Mappings_{final} can be found at our project website.

3.2 Violation Detection

3.2.1 Privacy-Phrase Ontology

With the API mapping, for a given app, we can detect violations of its privacy policy by checking the APIs it invokes, against the phrases appearing in the privacy policy. For example, if an location-fetching API is invoked and the location data is sent out, but none of the phrases about location is mentioned in the privacy policy, the app actually violates its privacy policy. However, one common phenomena in natural language description is generalization, in which a more general phrase can be used to imply a number of subconcept of the phrase. For example, the phrase “technical information” may imply a wide range of technical data, while the phrase “device id” is more specific, but its concept is still covered by phrase “technical information”.

Although using a general phrase (e.g., “technical information”) to describe a specific privacy information (e.g., “device id”) still results in imprecision of the privacy policy, it should be differentiated from the cases where none of the relevant phrases are mentioned about the privacy information being collected. To differentiate the two scenarios, we created an ontology of privacy-related phrases to be used as a cross reference when finding related APIs. An ontology is a formal description of entities and their properties, relationships, and behaviors [18], and is described with formal languages such as OWL (based on Description Logic).

In our scenario, the ontology can be viewed as a hierarchy of concepts, which describes the conceptual subsuming relationship between phrases. For example, in Figure 4, “MAC address” is a decedent of “Device Information”, indicating that MAC address is a direct relationship from the natural language privacy policies to the natural language documents. A documentation-based mapping would provide a relationship based strictly on natural language.

Figure 4: Illustration of Phrase Ontology

3

http://sefm.cs.utsa.edu/android_policy
noted as suspicious API detection in presented in Figure 5. After that, we use information flow analysis to further check whether the data fetched though suspicious API invocations is sent to remote servers (will be introduced in Section 3.2.3). In process overview of ontology-based suspicious API detection in presented in Figure 5.

Consider the set of extracted phrases $P$, the phrase-API map $map$, and the phrase ontology $O$, we can divide invoked information-collecting APIs $A$ into two categories: described (denoted as $D$), and missing (denoted as $M$). The two categories are formally defined as below. In the formula, following the formalization of description logic, we use $\subseteq$ to define concept inclusion in the ontology, and for two phrases $a$ and $b$, $a \subseteq b$ means that $a$ is a subconcept of $b$ in the ontology.

$$O' = O \setminus \{\text{information}\}$$  \hspace{1cm} (4)

$$D = \{x | \exists ph, \text{sub} \in O', ph \in P \land \text{sub} \subseteq ph \land x \in map(sub)\}$$  \hspace{1cm} (5)

$$M = A - D$$  \hspace{1cm} (6)

Based on formula, an API $x$ belongs to the described category if the policy mentions a phrase $sub$ mapped to $x$, or mentions a phrase $ph$ that has a decedent phrase $sub$ mapped to $x$. Specific to our problem, since the root node (“information”) in our ontology is too general and not descriptive at all, we use $O' \setminus \{\text{information}\}$ when categorizing APIs. Actually, we can see that an API can be either explicitly mentioned by a phrase that directly maps to it, or inferred by a phrase that describes a more general concept. So we further define the explicitly allowed category of APIs (denoted as $E$) as follows, and the inferred category of APIs can be defined as $D - L$. It should be noted that, invoking an API that can only be inferred is also harmful to privacy protection, because users may not be capable to do the inference.

$$L = \{x | \exists ph \in P, x \in M(ph)\}$$  \hspace{1cm} (7)

We also Figure 4 to illustrate the categorization of APIs. The privacy policy of an app may mention collection of “Device Information”, but do not mention the details. In such a case, the API “getMacAddress()” will be considered as an inferred API, which means that the privacy policy fails to provide sufficient details about the collected information. If the privacy does not mention any phrases which are ancestors of “MAC Address” (i.e., “Device Information” and “Unique Identifier”), the API “getMacAddress()” will be considered as a missing API. With the categorization of APIs, detection of suspicious API invocations is straightforward. We simply analyze the bytecode of an app and identify all invocations of missing and inferred APIs.

If an app invokes an API in the missing category $M$ and sends the fetched information to remote servers, the API invocation is defined as a Strong Violation of the privacy policy, and if an app invokes an API in the inferred category $D - L$, the API invocation is defined as a Weak Violation. While a strong violation is obviously harmful to the users’ privacy protection, a weak violation is an indication of lacking sufficient details and can be used to guide the improvement of privacy policies.

### 3.2.3 Information Flow Analysis

As we defined above, an API invocation becomes a violation only if the information it fetches is sent to remote servers. Therefore, we need to further check the destination of the fetched data, and existing information flow analyses provide the technique to this goal. We also require a list of sink APIs that send information to remote servers.

It should be noted that our framework works with any information analysis and list of sink APIs for network data transfer. In our implementation, we leveraged FlowDroid [6], the state-of-art technique for Android information flow analysis to track the information flow. We also used the list of sink APIs for network data transfer in SUsI [27], a very complete list of source and sink API for Android System.

### 3.3 Ontology Generation

Ontology can be useful in our violation detection, but it is not easy to construct. To create our ontology, we used the privacy policy ontology presented by Breaux et al. [10] and aligned the phrases with this existing ontology. This ontology was selected to be the core ontology for our alignment, since to the best of our knowledge, there is no other reference to privacy policy ontologies.

We manually searched for the 38 overlapping phrases and their synonyms in the core ontology. We determined the set of overlapping concepts, synonyms, and unique concepts to the main phrase groups [12]. Since the core ontology was created for a different application, there are multiple intermediate classes which do not apply to current research goal. For example, we are not interested in third parties and the information flow between them which made us eliminate some classes like “parties” or “technology” in mapping. While we are looking for the information collected by applications, we only consider “GPS” as subcategory of “Information” and we eliminate “GPS” as subcategory of “Technology” in our ontology. We also encountered some other differences and contradictions with the core ontology presented by Breaux et al. [10] during alignment. Some phrases like “MAC” and “MAC address” are decided to be equivalent by our research team.

The immediate sub-classes of the “Information” class that are identified through this phase are: “Communication”, “Identifiable Information”, “Listing”, “Location Information”, “Location Data”, “Network Information”, “Phone Information”, “Technical Information”, and “Unique Information”. Using the ontology generation tool Protege, we were able to classify the phrases and intermediate classes. Some phrases like “MAC address” are subsumed by two classes. Moreover, we can define synonyms by means of equivalent classes in Protege. For example, in the first level, “Location
Data” and “Location Information” are both in the 38 phrase groups and considered to be equivalent. The final ontology we generated is available at our project website⁴.

4. EVALUATION AND RESULT STUDY

In this section, we present an empirical evaluation of our framework by applying it to top Android apps and their privacy policies. In the evaluation, we try to answer the following research questions.

- **RQ1**: Is our framework able to detect violations of privacy policies in real-world Android apps?
- **RQ2**: What is the effectiveness of the different variants of our framework?
- **RQ3**: What are the major types of privacy information that are leaked in detected privacy-policy violations?

4.1 Study Setup

In this subsection, we introduce how we construct the data set for empirical evaluation, the metrics used, and the compared variants of our framework.

4.1.1 Data Collection

The first step in our evaluation is to construct a data set of Android apps with their corresponding privacy policies. In particular, from the official Google Play market, we downloaded the top 300 free apps⁵, as well as the top 20 free apps for each app category.⁶ We combined all the downloaded apps and acquired an app data set with 1,096 apps. Note that, although most Android apps have privacy policies, the app owners may put the policy at different places, such as their portal site at Google Play market, or a link in the main page of their company / organization. The privacy policy can also be in different formats, such as HTML, PDF, or Windows Word Document. Based on our observation, a large proportion of apps place their corresponding privacy policy at their portal websites at the Google Play market. Therefore, we crawled these websites and tried to automatically download the privacy policies of these apps. Furthermore, we considered only HTML privacy policies (the most popular format of privacy policies) in our evaluation for simplicity and avoiding potential noises in text extraction from various file formats. Note that, with proper text extraction tools, our framework can be applied to any format of privacy policies. Based on the automatic downloading and file format filtering, we collected privacy policies for 501 apps, and generated a data set with 501 apps and their corresponding privacy policies⁷.

4.1.2 Evaluation Metrics

To answer research questions **RQ1** and **RQ2**, we need to measure the effectiveness of violation detection. In our study, we leveraged the standard precision (what proportion of detected violations are true violations) and recall (what proportion of true violations are detected) as the measurement. It should be noted that, when counting violations in our study, we always count all invocation of the same API in an app as one violation.

We determined whether a detected violation is a true violation by manual inspection. In particular, each violation was assigned to and inspected by two of the authors, and a third author served as a judge if the two inspectors cannot get consensus. Another complication in the measurement is that, the ground-truth number of true violations in our whole data set is not known, and thus the recall value can not be calculated. Note that the manual inspection process is rather time-consuming and can hardly be applied to the whole data set which has 501 app-policy pairs and numerous suspicious API invocations. Therefore, in our study, we leverage the widely-used relative recall [20], which is a variant of the recall metric when the ground-truth number is unknown. Specifically, we use the union of true violations detected by all techniques as the ground-truth set of true violations, and calculate the relative recall of each technique based on it. Thus, the relative recall of a technique will be lower than its recall, but we can still use relative recall to compare different techniques.

4.1.3 Variant Techniques

In our empirical evaluation, to answer research question **RQ2**, we considered two variants of our framework, and compared them with our default technique. In the first variant, we leverage S/S I API categorization to filter our API mapping. In our default API mapping technique, we combine the knowledge from Android official document and the association between API access and phrases in privacy policies, but we did not use the S/S I API categorization. To further check whether the API categorization helps our task, we manually assigned phrases in our ontology to their most relevant S/S I categories (e.g., mapping “country” and “GPS” to the location category), and maps a phrase to only the APIs in its S/S I category (i.e., only APIs in S/S I Location category can be mapped to “country” or “GPS”). This variant is referred as “S/S I -filter” in the following parts of the paper. The second variant is about what phrases we choose to directly map to APIs. We may either choose to map only the leaf phrases (phrases with precise meaning and has no subconcept) in our ontology to APIs or all phrases. In our default technique, we map only the leaf phrases to APIs. In our second variant, we use the same API-mapping technique to map all phrases in the ontology to APIs, and refer to this variant as “all-phrases”.

4.2 Study Results

**Overall Results.** We applied our default technique to the 501 pairs of apps and privacy policies in our data set. For the 501 apps, with 30 minutes time-out for each app, FlowDroid successfully processed 397 of them. From these 397 apps, our default technique detected 85 violations in total, including 34 strong violations and 51 weak violations. Our manual inspection reveals that, among these detected violations, 63 are true violations, including 30 strong violations and 33 weak violations. The detection of 33 true weak violations shows that, our privacy-phrase ontology is helpful on differentiating weak violations from strong violations.

We further studied how these violations distribute among the apps, and the result is shown in Figure 6. From the figure, we can see that, we detected these violations in 31 apps, and most apps have under 3 violations. Actually, the 30 true strong violations are from 18 apps, and the 33 true weak violations are from 15 apps. To sum up, our default technique is able to detect privacy-policy violations in a significant number of top Android apps, and our false positive rate is relatively low so that developers do not need to waste too much effort on inspecting false violations.

**Study on detection errors.** Our default technique generates 4 false positives for strong violation and 18 false positives for weak violations.⁸

⁴http://sefm.cs.utsa.edu/android_policy
⁵The ranking of top 300 free apps is available at the Google Play website and was fetched on May 19th.
⁶The list of app categories is available at the Google Play website, and the top 20 apps for each category is fetched on May 19th.
⁷The data set is available at our project website: http://sefm.cs.utsa.edu/android_policy.
⁸Processing of the rest 104 failed due to time-out, heap overflow, or other exceptions.
From our manual inspection, the cause of the majority of the false positives is that, the privacy policy uses a phrase not in our ontology to describe the privacy information they collect. For example, one of the privacy policies uses the phrase “coordination” to describe gps information. Since the phrase is not in our ontology, our technique mistakenly determined that location information is not mentioned in the privacy policy and reported a false violation. Our default technique also missed 7 known strong violations and 2 known weak violations. Our manual inspection shows that the major reason is that the privacy policies use the privacy phrases for other purposes. For example, the phrase “geo location” may appear in the privacy policy, but instead of being used in the description of data collection, it may be used in a sentence talking about different terms and rules in different “geo location areas”. In such a scenario, our technique will mistakenly determine that the policy mentions collection of geo location information. From these observations, we believe that, applying more advanced natural language processing techniques in the violation detection phase may further enhance the effectiveness of our framework.

Comparison of variant techniques. To answer RQ2, we implemented the two variant techniques and applied them to our data set. The three techniques detected 100 violations in total (we can not simply break the number to strong and weak violations because different techniques may report the same violation as of different types), among which 37 are true strong violations, and 35 are true weak violations. Based on this ground truth, we calculated the precision and relative recall of each technique, and presented them in Table 4. In the table, columns 3-6 presents the number of detected violations, the number of true violations, the precision, and recall for each technique on each type of violations, respectively.

From the table, the following observations. First, our default techniques achieves relatively lower precision, but much higher relative recall, which means discovering more violations. Therefore, our default technique is in general more effective. Second, the S/St-filter technique achieves a very high precision and very low relative recall. With inspection of the missed violations, we found the major reason is that, one API can be mapped to multiple S/St categories. For example, “getNetworkOperatorName” can be mapped to both “network carrier” and “country”. Therefore, filtering API maps with S/St categories will actually remove some correct API-mappings, and result in missing true privacy-policy violations. Second, the all-phrases technique also achieves a high precision, and lower relative recall for weak violations. The reason is that, once the non-leaf phrases are directly mapped to the APIs, the mapping itself will become imprecise. For example, the phrase “location” may be mapped to GPS-based locations APIs, cell-station-based location APIs, as well as carrier-network-based or time-zone-based APIs. Therefore, if an app mentioned collecting “location” in its privacy policy, and sends GPS-based location information to network servers, it will not be viewed as a violation, while it is actually a weak violation.

**Top privacy information types in detected violations.** To understand what types of privacy information are involved in privacy-policy violations, we studied the 5 APIs and phrases that are associated with most of our detected true violations, and the results are presented in Figure 7, and Figure 8. In the two figures, the y-axis shows the name of the API or phrase, and the x-axis shows the number of strong (shown in gray) violations and weak (shown in black) violations associated with the API or phrase. For brevity, we present only the short name of APIs in Figure 7.

From Figure 7, we can observe that the top APIs associated with detected violations are all related to location information. In particular, the top 3 APIs “getLastKnownLocation”, “getLatitude”, and “getLongitude” request the precise GPS information of a device, and the 4th and 5th APIs, “getNetworkOperatorName” and “getTime” can be used to infer coarse-grain location information. From Figure 8, we have a similar observation that location information is the most important type of information involved in detected violations, and “Network Operator”, and “Device Identifier” are two other major types of information. It should be noted that, location information is of great importance compared to other types privacy information, and collection of location information is specially required to be explicitly stated in the privacy policies [1].

**4.3 Threats to Validity**

**4.3.1 Internal Validity**

The major threat to the internal validity of our empirical evaluation is the correctness of our ground-truth violations. It is possible
that we make mistakes on identify true violations. To reduce this threat, we have at least two people working on each violation, and we also carefully checked the texts in the privacy policy and the information-flow path detected by FlowDroid.

Ideally, we would have been able to confirm violations with developers. We attempted to contact 5 projects with explicit violations detected about their violations via email but have not yet received any responses. This is likely due to the fact that developers and publishers do not want to risk the legal ramifications of admitting to breaking their own privacy policies.

4.3.2 External Validity
The major threat to the external validity of our empirical evaluation is that our results may apply only to the apps used in the study. To reduce this threat, we collected top apps from all different app categories to enhance the representativeness of our data set. Also, it is possible that our results apply to only top Android apps, and to further reduce the threat, we plan to apply our framework to other types of application, such as IOS-based apps and web applications.

5. DISCUSSION
5.1 Violation Detection
We have shown in Section 4.2 that the mapping can facilitate policy violation detection with relatively high precision and recall. This trait makes the data useful for those who may not have the necessary knowledge to verify privacy policies themselves. Such a mapping could be used to assist developers in detecting potential violations. In such a scenario, the tool could bring to light potential violations between policy and code that may not be immediately or intuitively obvious to the user.

Violation detection is improved through the use of our ontology (as relevant in the comparison of explicit and implicit violations). The ability for compliance to be implied through transitive relationships between terms allows for the improvement of the overall approach without the need for more app code and privacy policies. This aspect of our work has the potential to be applied to new and different apps. We plan to apply our framework to other types of application, such as IOS-based apps and web applications.

6. RELATED WORK
Prior work exists on the factoring of privacy and privacy policies into source code. To our knowledge, ours is the only technique that works to bridge the gap between privacy policy and implementation through the use of natural language mappings to APIs.

6.1 Privacy and Permissions
Android has a permission system that is used for apps to go gain access to certain APIs. An app must declare these permissions as part of its source code. In turn, the user is notified during installation as to what the app requests. This permission system is related to privacy policies in that the APIs that are accessed through the permissions are, or should be, represented in the app’s privacy policy. An app’s privacy policy can be cross-checked with an app’s permissions, but permissions are not necessarily defined at API-
level granularity. There is existing research that explores both privacy policies and OS permissions.

Rowen et al. have developed an IDE plugin, Privacy Policy Auto-Generation in Eclipse (PAGE), for generating privacy policies along side the development of the app [29]. PAGE works by guiding the user through a series of questions about the implementation of the app. Based on the answers, PAGE uses existing policy templates to generate a privacy policy for the app. Unlike our technique, PAGE does not take into account API calls or information flow and cannot be used for the detection of policy violations. As described in Section 5.1, we are working on a tool that uses our mappings to both generate natural language phrases for use in privacy policies and verify the accuracy of a policy with respect to its app.

The static analysis tool PScout was developed by Au et al. for the analysis of the Android OS permission system [7]. The Android permission system helps policy consistency since it is used to show, at a course-grained level, the data that the app can access. PScout maps these permissions to API calls in order to evaluate their coverage. PScout itself does not work as a tool for linking policies to code, but its analysis of the Android permission system shows a limited interconnection of API-permission mappings. Among our mappings, only 22% of the APIs only mapped to one phrase. We believe this is due to the empirical aspect of our approach. By including the frequency at which APIs appeared in real-world apps with regard to phrases in their policies, it is more likely for a phrase to correspond to more than one API. This effectively adds more data to the violation detection process.

Existing work by Petronella presents a tool that relates natural language in privacy policies to Android permissions [26]. The tool works by providing the user with the list of permissions for an app along with the sentences from the privacy policy that are related to each permission. This is similar to our work in that it maps natural language phrases to potential data collection actions in the app’s implementation. However, it is limited to the granularity of the Android permission system. Our work looks past permissions and related policy phrases directly to API invocations. Furthermore, our phrases are mapped using empirical and documentation information whereas the relationships between phrases and permissions for Petronella’s tool were manually associated based on the discretion of the author.

Stoaway, a tool created by Felt et. al., detects over-privilege in compiled Android apps [14]. The tool found that among 940 apps, 35.8% were overprivileged. Analysis of the apps showed that copied code and testing artifacts were among the reasons for unnecessary privileges. These kinds of defects can be brought to light with policy verification through the use of API mappings.

Vidas et al. have created a tool, Permission Check Tool, to assist developers in selecting minimal necessary permissions [32]. The tool uses static analysis to check the code for API references. Those references are cross-checked with a permission-API database created by the authors in order to generate a minimal set of permissions. These tools help to minimize privacy issues purely from an implementation standpoint. Our mappings can be used in conjunction with these to verify the corresponding policies.

Similarly, Bello-Ogunu et al. have created an Eclipse IDE plugin, PERMITMe to guide developers in selecting Android permissions [8]. The authors tested the tool on students in a mobile application development course and found that the assistance reduced the time spent on assessing privacy permissions and was found to be helpful and welcomed overall. PERMITMe does not integrate natural language privacy policies in any way. However, the results of their study support the idea a privacy tool targeted at developers may help to alleviate the time spent on privacy compliance. Our data would allow for more privacy assurance with policy checking.

A new model is being used for the permission system in the upcoming Android 6.0 [4] While the new model is backwards compatible with apps that use the old model, new apps will use new features that may affect previous work regarding the old permission model. Our framework is more robust in the sense that it is not affected by changes to the permission model.

6.2 Security and Privacy Ontologies

Many research has been carried out on ontology implementation and usage in computer and information systems in recent years [9, 11, 16, 21, 31]. However, the main concentration of these works are using ontology for permission based systems, firewalls and pervasive systems. These ontologies are based on policies defined by system administrators or system clients/users. The only ontology which considers privacy policy of applications is the one presented by Breaux et al. [10]. This ontology considers privacy policies for web-base and mobile applications in different domains of data and information collection, usage, transfer to third parties, and retention. However, in our ontology we are considering Android mobile applications without taking web-based applications into account and our focus is on data and information collected by Android Apps. Therefore, there was a need to generate a new ontology which fulfills our requirements. In this section, we briefly discuss the previous works on ontology implementation and usage and their differences with our main goal of research.

Breaux et al.[10] implemented an ontology to analyze the privacy policy of multi-tier systems to find the conflicts between the policies regarding data collection, usage, retention and transfer. For this reason, the authors consider Facebook, Zynga (company which developed the Farmville) and AOL advertising company which provides advertisements in Farmville. To identify the conflicts between privacy policies and ensure the consistency across different parties which are defined in natural language, Breaux et al. tries to extract the privacy requirements and then map the requirements to description logic to minimize the ambiguity related to the words and statements in natural language using privacy policy ontology implemented by them. Finally, the authors consider the conflicts in each privacy policies separately along with the conflict between Facebook and Zynga, and Zynga and AOL as direct information flows.

In other research, Chen et al.[11] implemented a shared ontology for ubiquitous and pervasive applications called SOUPA. Their main aim was pervasive systems and included modular component vocabularies to represent intelligent agents with associated beliefs, desires, and intentions, time, space, events, user profiles, actions, and policies for security and privacy [11]. This research take advantage of spatial relations in reasoning about the collected information from user. They used the shared ontology to facilitate decision making in dynamic environment. This ontology does not consider the privacy policy related to the mobile applications and is mainly concerned about security policies in pervasive systems and knowledge sharing using the ontology.

In the domain of ontology as access control and permission systems, Kagal and Finin[21] considered client-server model and the privacy policies related to both web services and clients as users of web services. The ontology implemented by them was used as access control model. This ontology was implemented based on user specification and requirements of web services, the information that users agreed on sharing with web services and web services specification. As we can see, the main focus of this research is also web-based applications, user and system policies to control
the system permissions.

Bradshaw et al. [9] implemented an ontology based on policies for behavior regulation of intelligent agents to continually adjust their behavior and maximize the effectiveness in intelligent systems. The ontology is used for specification, management, conflict resolution, and enforcement of policies in intelligent systems which does not align with our goal in mobile applications privacy policies.

Finally, in access control domain, Grandon and Sadeh [6] tried to preserve users' clients contextual information from being misused by web-applications using ontology as access control mechanism. In this research, users can control who has access to their contextual information and under which conditions. Therefore, they only consider user policy and try to align the policy with the web-based applications.

6.3 Information Flow Analysis

In our work, we leverage information flow analysis to check whether privacy information accessed from a certain API flows to the network, and we used FlowDroid [6], a state-of-art tool for static information analysis of Android apps. Other Android-oriented static information analysis techniques include CHEX [23], LeakMiner [33], and ScanDroid [15]. Specifically, CHEX detects component hijacking vulnerabilities in Android applications by tracking taints between externally accessible interfaces and sensitive sources or sinks. LeakMiner is an earlier context-insensitive information-flow analysis technique for detecting privacy leaks in Android apps. ScanDroid tracks information flows among multiple apps and detects privacy leak into other apps. There are also dynamic information flow analysis techniques such as TaintDroid [13] and CopperDroid [30] that may detect privacy leak at runtime. It should be noted that all of the information-flow analysis techniques take a formal specification (e.g., a list of allowed flows from certain source APIs to sink APIs) and detect privacy leaking. However, the privacy policies are written in natural language so information-flow analysis cannot be directly applied. Our framework helps to bridge the gap by providing the ontology and the API mappings, and may benefit from more advanced information flow analysis techniques in the future.

7. CONCLUSION AND FUTURE WORK

Smartphone applications collect significant amount of privacy sensitive data from their users’ devices. Organizations that develop these applications have a legal and moral obligation to clearly articulate to their users what data are being collected in the form of an application-specific privacy policy. However, these organizations lack sound mechanisms that can help them determine if the stated privacy policy is accurate—i.e., are the applications collecting pieces of privacy sensitive data that is not stated in the policy? This paper has made significant strides toward addressing this problem by developing a framework for detecting privacy policy violation in Android application code.

At the same time, our work in this paper has revealed a problem that is much more broader in scope. For instance, our framework in this paper helps detect if an Android device-related data collection API is invoked, and if the data is transmitted through the network by the application code. This does not necessarily imply that the data is eventually stored (i.e., collected) in the organization’s servers. To determine if the data that is written to the network is actually collected and stored, the flow of information needs to be traced from the user’s phone to the respective organization’s data repositories. This requires an organization to analyze potentially disparate applications, web frameworks, databases, etc. that could span multiple systems. Furthermore, collection of data does not inform how the data is used and shared by the organization. To validate if application code, and data use and sharing aspects of the privacy policy are consistent, new techniques need to be developed. Our future research will proceed along these avenues.

8. REFERENCES


