

PoliDroid-AS: A Privacy Policy Alignment Plugin for Android Studio

Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, Jianwei Niu
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas, USA
{rocky.slavin,xiaoyin.wang,mitra.bokaeihosseini,jianwei.niu}@utsa.edu

Jaspreet Bhatia, Travis D. Breaux
Institute for Software Research
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
{jbhatia,breaux}@cs.cmu.edu

Abstract—Mobile applications frequently access personal information to meet user or business requirements. Developers are, in turn, often required to align their code with a privacy policy or to create the privacy requirements to specify the collection and use of personal information. However, it is challenging for a regular programmer to create code complying with a privacy policy. To aid app developers in such tasks, we have created PoliDroid-AS, an Android Studio plugin for the detection of code-policy misalignments and the generation of privacy specifications.

Keywords—mobile; privacy; Android;

I. INTRODUCTION

Due to the nature of mobile devices, mobile applications (apps) have the potential to access various sensors and private information stored on the device. Such data access is not necessarily nefarious (e.g., it is not unreasonable for a navigation application to access the device’s GPS coordinates), however the Federal Trade Commission suggests application publishers disclose the types of information that are collected so that users can be aware of what private information may be shared with the app. Further, end users are becoming more aware of the ramifications of private data collection [5]. For these reasons, it is important that software developers not only be aware of what information their app is collecting, but whether or not their privacy policy is consistent with what is actually being collected or accessed.

Privacy policies are not typically written by the developers themselves. For larger companies, this task is commonly assigned to legal experts since the policy is a legal document. These experts may not be familiar with the actual app code. It is then the developer’s job to create code that adheres to the policy. This job can be difficult since the relationships between natural language and source code are not always obvious depending on the writers, developers, and the (sometimes intentional) vagueness of legal jargon. For smaller companies that cannot afford legal expertise, the developers end up with the task of writing the policies, leading to copied or misaligned policies.

To aid developers in both aligning their code with privacy policies as well as generating privacy specifications, we present PoliDroid-AS, an Android Studio plugin which uses our previous work [8] on privacy policy violation detection which visually notifies the developer of misalignments in real-time as they write code. PoliDroid-AS is a part of a larger suite of privacy-related tools for Android. Other tools [7], [3], [2], [1], [9] attempt to aid in such tasks, however, to our knowledge, ours is the first to leverage natural language processing and a privacy ontology to directly aid a developer in writing code. Furthermore, we present the results of a case study of the plugin’s usage on the development of real-world apps.

II. COMPONENTS AND ARCHITECTURE

As seen in Figure 1, PoliDroid-AS is built upon a collection of tools that allow it to utilize our privacy policy violation detection framework within Android Studio, the predominant IDE for Android development.

A. User Input and Supporting Artifacts

In order to understand the plugin’s architecture, it is necessary to know its possible inputs.

1) Android Application: PoliDroid-AS operates on one application at a time and thus takes as inputs the application’s privacy policy and its source code.

Since privacy policies are generally included along with other legal documents (e.g., terms of service) often displayed on the application’s web page or Google Play, PoliDroid-AS is designed to take as input either a plaintext document or an HTML file including the policy. The relevant privacy policy sections are automatically extracted from the entire document (see Section II-B). US regulators require that privacy policies be in English, so the tool also requires an English document. The following is an abbreviated example of an HTML input that includes both privacy policy text and irrelevant terms of service text. Information relevant used in our running example is bolded:

```
<h3>Information We Collect</h3>
<p>When you use our service, we may collect info such as your mac address and address.</p>
<h3>Terms</h3>
<p>By using our app, you agree to the following ...</p>
```

1http://polidroid.org
PoliDroid-AS allows for real-time consistency verification, thus the app’s source code can be scanned “on-the-fly” as the app is being developed. The tool expects Java source code as the relevant code used for accessing private information.

2) Privacy Policy Violation Detection Framework: For our prior work on a privacy policy violation detection [8], we created a set of many-to-many mappings and a mobile privacy ontology used to identify the relationships between Android application program interface (API) method calls and natural language phrases pertaining to information collection specifically from three predefined [6] categories: network, location, and unique identifier information. The mapping directly relates API methods to phrases that can exist in the privacy policy (see Table I for example). The ontology allows the framework to indirectly relate phrases to API methods for which no mappings exist. If an API method is represented in a privacy policy through an indirect mapping (i.e., the phrase in the policy subsumes a phrase to which the method is mapped), the instance is reported as a weak violation. If no mapping (direct or indirect) exists, it is reported as a strong violation.

B. Privacy Policy Processing

Before code analysis, an app’s privacy policy must be preprocessed to extract the relevant information.

If the input is in HTML form, PoliDroid-AS calls upon an existing HTML parsing Java library, Jsoup2, to strip the HTML tags and produce a plaintext version of the input. The plugin then filters the text down to only privacy-relevant data collection paragraphs. Next, all sentences in the resulting plain text are converted to relevant parse trees using Stanford CoreNLP3 parser. For each parse tree, the verb phrases (VPs) are extracted and if the lemmatized verbs of a VP contains a collection verb (e.g., “store”, “collect”, “record”, etc), the VP is selected for more analysis. The reduced constituents are compared with the mobile privacy ontology concepts to find a match using the Web-Ontology Language (OWL) API. PoliDroid-AS then searches for phrases in the resulting paragraphs for which API methods are mapped, as mentioned in Section II-A2.

C. JetBrains IDE SDK

The Android Studio IDE is based on the IntelliJ IDEA Java IDE by JetBrains. PoliDroid-AS was built using the JetBrains IDE Plugin SDK4 so that it can directly interface with IntelliJ-based IDEs. PoliDroid-AS’s core component is an extension of IntelliJ’s inspection tool which allows PoliDroid-AS to visit all API method calls within the source code being developed within the IDE and verify that the API method call is represented by the app’s privacy policy.

III. PLUGIN FUNCTIONALITY

After providing the tool with the API-phase mappings, ontology, and privacy policy, the PoliDroid-AS Android Studio plugin works seamlessly with Android Studio without further configuration and minimal interaction.

A. Set Up

Once the app developer has started Android Studio, they will be presented with a “PoliDroid” menu in the IDE’s menu bar. Clicking this menu will result in four options: “Select Mappings File”, “Select Ontology File”, “Select Policy File”, and “Generate Specifications”. The first three of these selections prompts the user to load the corresponding file. The mappings file is a simple comma-separated values (CSV) file including the many-to-many mappings of APIs to privacy information collection phrases. This mapping is included with the plugin. The ontology file is an OWL formatted document specifying the subsumption, equivalence, and sibling relationships used for identifying weak and strong violations. Like the mappings file, a pre-defined ontology is included with the tool. We provide the option to specify custom files for the sake of flexibility.

Once the framework files have been loaded, the app’s privacy policy can be specified from the same menu. PoliDroid-AS immediately strips the policy of any html markup and extracts the relevant privacy policy paragraphs (as described in Section II-B) immediately upon loading the file.

---

2http://jsoup.org
3http://nlp.stanford.edu/software/corenlp.sHTML
4http://www.jetbrains.org/intelliJ/sdk/docs
Next, PoliDroid-AS generates a lookup table of API methods represented in the privacy policy through the mapping \( A_{\text{represented}} \) by searching through the policy text for all mapped phrases. A set of omitted API methods is then generated as \( A_{\text{omitted}} = A_{\text{mapped}} \setminus A_{\text{represented}} \) where \( A_{\text{mapped}} \) represents the set of all methods for which the mapping file has phrases mapped to. \( A_{\text{mapped}} \) must be used as opposed to the set of all API methods because the tool cannot make assumptions on methods for which it has no data.

Continuing with our running example, the phrases “mac address” and “address” existing in the source along with the mappings from Table I would result in \( A_{\text{omitted}} = \{ \text{getIP}() \} \) since the other method represented in \( A_{\text{mapped}}, \text{getMac}() \) is represented by the phrase “mac address” in the privacy policy.

B. Detection and Reporting

After the generation of \( A_{\text{omitted}} \), PoliDroid-AS can then scan for API method calls not represented in the policy by checking for any occurrences of methods in \( A_{\text{omitted}} \).

PoliDroid-AS utilizes the IntelliJ LocalInspectionTool class to inspect all method calls within the development code. The implementation is able to match API methods by comparing a method’s signature, including the method name, class, and package. If, upon inspecting a method, PoliDroid-AS detects a violation \( \omega \in A_{\text{omitted}} \), it registers the violating method call and highlights the offending method call directly in the IDE as seen in Figure 2 (the annotation style can be modified to the user’s preference within Android Studio’s settings menu). In our example, a call to getIP() would be a member of \( A_{\text{omitted}} \) (i.e., not be represented in the privacy policy) and thus be flagged as a potential violation.

To determine information about the violation, the offending API method, \( \omega \), is cross-referenced with \( A_{\text{mapped}} \) to determine all phrases to which it is mapped, \( \Phi_\omega \). The mobile privacy ontology is then used to determine the set of hierarchical ancestors, \( \Psi_\omega \), of all \( \phi \in \Phi_\omega \). These phrases semantically subsume at least one member of \( \Phi_\omega \) so that \( \Psi_\omega = \{ \psi \mid \phi \in \Phi_\omega \land \phi \sqsubseteq \psi \} \) according to the ontology. The members of \( \Psi_\omega \) represent indirect, or transitive, representations of \( \omega \) since they are not directly mapped to the \( \omega \), but are related through a subsumption relationship. For this reason, \( \psi \in \Psi_\omega \) can be checked for matches in the privacy policy to determine if the violation is considered weak or strong. If any phrase in \( \Psi_\omega \) is found in the policy, the violation is flagged as weak (i.e., it does not have a phrase directly mapped to an API method, but the method is represented through a transitive relationship in the ontology), otherwise it is flagged as a strong violation (i.e., there is no relationship between any phrase in the ontology with the offending method call).

For our running example, \( \omega = \text{getIP}(), \Phi_\omega = \{ \text{“ip address”}, \text{“unique identifier”} \} \). Based on the sample ontology in Figure 3, \( \Psi_\omega = \{ \text{“identifiable information”}, \text{“address”}, \text{“technical information”} \} \). Since the phrase “address” appears in the privacy policy, \( \omega \) can be reported as a weak violation even though its mapped phrases, \( \Phi_\omega \), do not appear in the policy. This ability for the ontology to allow transitivity-related phrases to represent methods helps to reduce false-positives by accounting for phrases with broader meanings.

Hovering over the highlighted code will reveal a tooltip including information about the violation (see Figure 2). Sample phrases \( \Phi_\omega \) are suggested for insertion into the privacy policy in order to address the violation along with an indication of a weak or strong violation.

C. Specification Generation

PoliDroid-AS is also equipped with a privacy specification generator which reverse engineers a set of privacy specifications based on API invocations in the app’s source code. The generator works by presenting the developer with a wizard that
iterates through each invocation to an API method which has privacy phrase mappings. For each method, the developer is given a menu (Figure 4) that allows them to select or describe various details regarding the reasoning for using the privacy-sensitive method. Details include purpose, action verb (e.g., collection, sharing, etc.), information type, if the data will be shared with third parties, and purpose of sharing. Once the questions have been completed, they are compiled into a textual representation of the privacy requirements (see below). In turn, these requirements can be used to construct a privacy policy for the app.

<table>
<thead>
<tr>
<th>gps is COLLECTED with the following specifications:</th>
</tr>
</thead>
<tbody>
<tr>
<td># The gps data used is for the app’s basic functionality.</td>
</tr>
<tr>
<td># The gps data used is for business reasons.</td>
</tr>
<tr>
<td># The gps data will be used for: to determine where you are</td>
</tr>
<tr>
<td># The gps data will be stored off the device on servers.</td>
</tr>
<tr>
<td># The gps data will be stored for 1 week on such servers.</td>
</tr>
<tr>
<td># The gps data will NOT be shared with third parties</td>
</tr>
<tr>
<td># Such third parties will use the gps data for: NOT SPECIFIED</td>
</tr>
<tr>
<td># The gps data will be accessed by NOT SPECIFIED within the organization.</td>
</tr>
</tbody>
</table>

### IV. Case Study

We recruited seven developers working on their own pre-existing Android apps in Android Studio to conduct a case study on the usage of PoliDroid-AS’s violation detection and specification generation. The participants included graduate and undergraduate computer science students and a computer science faculty member. All participants had experience with Android Studio.

Prior to using our tool, we had the participants undergo a training session for the creation of privacy policies based on the California attorney general’s guidelines for privacy for the mobile ecosystem [4]. Participants were then asked to create a privacy policy for their apps.

Participants tested the violation detection feature of the plugin on computers pre-loaded with Android Studio and PoliDroid-AS. They were asked to import their code and privacy policies, open each class file, and report any violations detected by the plugin. Table II shows the results. Three of the participants found violations using our plugin. The violations all fell into the categories of location and network information. It interesting to note that each participant addressed their violations by updating their policy as opposed to the code.

Groups two and five reported that their apps simply did not collect sensitive information. We consider the data from these groups to still be relevant in that they did not exhibit false positives. Participant seven’s code did not exhibit violations due to the fact that the plugin could not detect invocations from within compiled libraries. We found that participant three’s code called upon the Build.MODEL field, a field containing the device’s model, which was not detected. This is a limitation of the framework, which only considers methods, and not the plugin itself.

Participants were also asked to use the privacy specification generation feature of the plugin to create specifications for their apps. They were asked to then annotate their original privacy policies noting if and where the information in the specifications was included. The results are seen in table III. As described above, groups two, three, five, and seven did not report any violations. For the remaining groups, an average of 55.2% of the information generated by the specification generator was not included in the participants’ privacy policies. This shows that specifications generated by the tool would help in reducing omissions in policies.

### V. Conclusion

Policy-code consistency is an important aspect of app development that can often be left up to developers who have little experience or interest in the legalities involved. Our tool allows such developers to produce and verify these documents in a practical way by integrating our framework with a familiar environment. The results of our case study demonstrate this and, furthermore, show an improvement in privacy disclosure coverage when compared with manually-created policies.

### References


