PAcT: Detecting and Classifying Privacy Behavior of Android Applications

Vijayanta Jain
University of Maine, Orono
Orono, ME, USA
vijayanta.jain@maine.edu

Sanonda Datta Gupta
University of Maine, Orono
Orono, ME, USA
sanonda.gupta@maine.edu

Sepideh Ghanavati
University of Maine, Orono
Orono, ME, USA
sepideh.ghanavati@maine.edu

Sai Teja Peddinti
Google Inc.
Mountain View, CA, USA
psaiteja@google.com

Collin McMillan
University of Notre Dame
Notre Dame, IN, USA
cmc@nd.com

ABSTRACT

Interpreting and describing mobile applications' privacy behaviors to ensure creating consistent and accurate privacy notices is a challenging task for developers. Traditional approaches to creating privacy notices are based on predefined templates or questionnaires and do not rely on any traceable behaviors in code which may result in inconsistent and inaccurate notices. In this paper, we present an automated approach to detect privacy behaviors in code of Android applications. We develop **P**rivacy **Ac**tion **T**axonomy (PAcT), which includes labels for *Practice* (i.e. *how* applications use personal information) and *Purpose* (i.e. *why*). We annotate ∼5,200 code segments based on the labels and create a multi-label multi-class dataset with ∼14,000 labels. We develop and train deep learning models to classify code segments. We achieve the highest F-1 scores across all label types of 79.62% and 79.02% for *Practice* and *Purpose*.

CCS CONCEPTS

• **Security and privacy → Privacy protections**; • **Computing methodologies → Natural language processing**.

KEYWORDS

privacy-behavior, privacy notices, Android applications, inconsistency analysis

ACM Reference Format:
Vijayanta Jain, Sanonda Datta Gupta, Sepideh Ghanavati, Sai Teja Peddinti, and Collin McMillan. 2022. PAcT: Detecting and Classifying Privacy Behavior of Android Applications. In *WiSec '22: 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks, May 16–May 19, 2022, San Antonio, Texas, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3507657.3528543

1 INTRODUCTION

A "privacy notice" is an application's artifact that describes how and why that application uses personal information, such as for providing services or monetization [34]. Regulations, such as General Data Protection Regulation (GDPR) [7] and California's Consumer Privacy Act (CCPA) [9] require applications to provide users with privacy notices [1]. However, developers face several barriers in creating high-quality privacy notices. First, developers often do not comprehend concepts of privacy well [10] and therefore, use privacy policy generators [47] that incorrectly capture privacy behaviors of their application. Second, they often use default configurations for third-party libraries without understanding their privacy behaviors and how it affects privacy of their applications [27]. Third, developers often do not consider privacy notices as software engineering artifacts and thus, they do not update them with source code [48]. Lastly, developers are not always included in the process of creating privacy notices; instead companies hire legal experts to ensure compliance which may result in mismatches between the application and its privacy notice [40]. These problems may result in creation of inconsistent privacy notices, which may lead to privacy harms for both users and developers.

Some work focus on resolving the above challenges by creating privacy notices using questionnaires or predefined templates [24, 37, 38, 45, 48]. While these approaches are promising, they generate generic notices without making them traceable to source code or matching them with applications' privacy behaviors. Without traceability, privacy notices become inconsistent when source code changes. In our previous work [15], we proposed a framework called *PriGen*, which identifies and extracts code segments that access personal information and then, uses a deep learning model to directly translate them into short descriptive privacy sentences called *privacy captions*. PriGen introduces traceability and achieves good accuracy for very short code segments (∼4-5 LOC), but the accuracy decreases significantly when code segments become larger. Additionally, training a deep learning model for caption generation requires a large high-quality dataset (≥20,000 samples), which is laborious and expensive to create manually. Furthermore, privacy is subjective; therefore, creating semantically similar privacy captions for the same source code is challenging, especially when there are multiple annotators.

In this paper, we develop an automated approach that can detect privacy behaviors of applications' source code which can later be used as input to *PriGen* to create the privacy captions dataset. First, we formally define categories of privacy behaviors in source code and develop ***P**rivacy **Ac**tion **T**axonomy* (**PAcT**). PAcT consists of two categories of labels: *Practice* and *Purpose*. The *Practice* category focuses on understanding *how a code segment uses personal information?* and *Purpose* category helps understand *why?*. This approach aides us in two ways: (1) defining privacy behavior categories alleviates any privacy subjectivity of annotators and makes labeling source code a structured process. It also enables us to create a dataset with consistent labels. (2) labeling code segments is a much simpler and cost-efficient task than writing privacy captions. Hence, we can manually create large datasets of code segments and their privacy behaviors. Second, we use PAcT and its guidelines to label privacy behaviors of ~5,200 code segments and create a large multi-label multi-class dataset with ~14,000 labels. For this labeling task, we developed a custom built open-source annotation tool, *Codr*. Finally, we develop and train a simple Recurrent Neural Network (RNN) model to classify privacy behaviors of Android applications. We conduct 32 experiments to examine how the unique characteristics of various *Practice* and *Purpose* labels as well as the amount of data extracted from source code can impact the model's performance. Our model achieves the highest F-1 scores of 79.62% and 79.02% across the label types for *Practice* and *Purpose* respectively (see Tables 2 and 3), which demonstrates the feasibility of classifying privacy behaviors in source code. While source code classification tasks were previously done for malware detection and other similar tasks showing very high performance, there is no prior work that detects *Practice* and *Purpose* from source code. Our work is the first to conduct such comprehensive analysis. Hence, our experimentation establishes the first baseline.

Our approach to detect privacy behaviors helps resolve the shortcomings of creating consistent privacy notices in the following ways: first, our trained models provide simple labels which help developers better understand privacy actions of their applications. For instance, a code segment accessing *location* and tagged with `Collecting` *Practice* and `Functionality` *Purpose* indicates that their application '`collects` *location data to provide* `functionality`'. In cases that developers create privacy policies via policy generators, they can use the predicted labels as an input to modify relevant sections of the generated policy to ensure the policy matches with the application's privacy behaviors. Additionally, since we create labels for all source code that access personal information, we mitigate the problem of missing privacy statements in policies [49]. Second, developers can use the generated labels to create intermediate notices. These intermediate notices can then be provided to legal experts writing privacy policies to help reduce the number of mismatches between application's privacy behaviors and notices. Appendix A - Figure A.1 presents a few more examples about how to create privacy notices using the generated labels. Lastly, since the labels are generated directly from source code, they can be traced as part of software engineering artifacts. Hence, any time the code gets updated, a new sets of labels will be generated which can result in updated or new intermediate privacy notices. This way the privacy notices will remain consistent with applications throughout their life cycle.

The main contributions of this work are as follows: (i) PAcT - a unique taxonomy to help detect privacy behavior of applications' source code. To the best of our knowledge, this is the first work to create such a taxonomy. PAcT bridges the linguistic gap between applications' source code and their respective privacy notices by creating a common vocabulary. This common vocabulary can then be used to describe privacy behavior of source code, and can later simplify the process of creating accurate and consistent privacy notices. (ii) Annotated Dataset of Privacy Actions (ADPAc) and Models - a dataset of ~5,200 code segments annotated with *Practice* and *Purpose* labels along with trained RNN models to classify them[1]. (iii) Codr - our open-source annotation tool to facilitate the creation of ADPAc dataset. Codr can be used by the research community to annotate code and other text for various tasks.

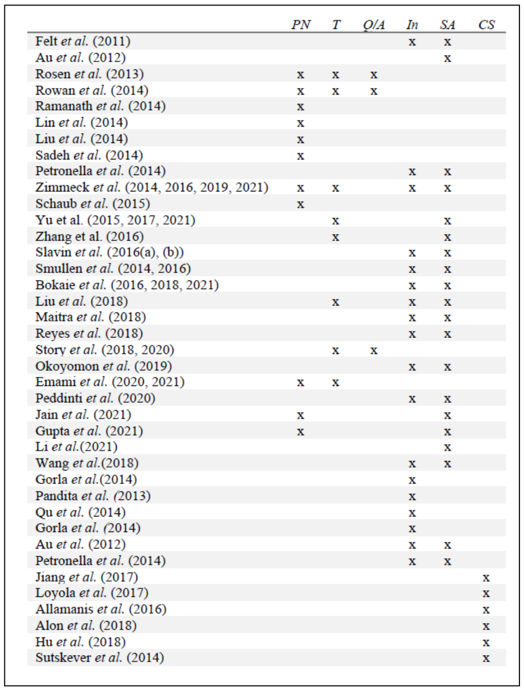
2 RELATED WORK

Figure 1 illustrates trends in the prior work. In recent years, much work have been done to help generate concise privacy notices for mobile applications by considering a set of questions or predefined templates [22, 37, 38, 45–47]. The first attempts, such as PAGE [38] and AppProfiler [37], concentrated on generating privacy policies from a set of questions, similar to most privacy generator tools. In the next attempt, some work focused on developing automated approaches or predefined templates to help write the rationales behind the required permissions, or create privacy notices or descriptions by analyzing sensitive APIs [45–47] and mapping them to natural language text. AutoPPG [45], PrivacyFlash Pro [47], and Honeysuckle [22] are all closely related to our work, and aim at creating privacy notices by analyzing code. AutoPPG uses a predefined `subject form object [condition]` format to create privacy statements, which results in generic statements that do not provide *Purpose* of privacy behavior. Both PrivacyFlash Pro and Honeysuckle provide rationales for using personal information, however, they both rely on developers to provide them. Our work differs from these approaches in that we classify privacy actions of source code and provide *Practice* and *Purpose* labels without using specific formats or relying on developers to annotate their code.

While these approaches attempt to generate privacy notices, they still lack traceability between the code and privacy notices which may result in inconsistencies between the applications' privacy policies/notices and code, especially when the code evolves. As such, some research focus primarily on evaluating the extent of such inconsistencies [36, 39, 40]. Some of these work analyze the discrepancies between applications' API calls and their privacy policies [11, 12, 23, 26, 28, 32, 39, 40, 43, 49], or with their applications' descriptions [8, 33, 35]. The main goal of these approaches is to identify violations and not to *mitigate and resolve* them. In contrast, our work supports traceability and helps resolve inconsistencies, since *Practice* and *Purpose* classifications in code can be directly translated to privacy notices.

Source code summarization is the task of automatically generating natural language descriptions of source code which ranges from creating commit messages and descriptions of source code changes [16, 25], predicting a method's name using a code snippet [2], to creating descriptions of a code segment [4, 13, 42]. The primary focus of most code summarization approaches is on writing high-level descriptions suitable for general-purpose programming

[1]https://github.com/PERC-Lab/PAcT



	PN	*T*	*Q/A*	*In*	*SA*	*CS*
Felt *et al.* (2011)				x	x	
Au *et al.* (2012)					x	
Rosen *et al.* (2013)	x	x	x			
Rowan *et al.* (2014)	x	x	x			
Ramanath *et al.* (2014)	x					
Lin *et al.* (2014)	x					
Liu *et al.* (2014)	x					
Sadeh *et al.* (2014)	x					
Petronella *et al.* (2014)				x	x	
Zimmeck *et al.* (2014, 2016, 2019, 2021)	x	x		x	x	
Schaub *et al.* (2015)	x					
Yu et al. (2015, 2017, 2021)		x			x	
Zhang et al. (2016)		x			x	
Slavin *et al.* (2016(a), (b))				x	x	
Smullen *et al.* (2014, 2016)				x	x	
Bokaie *et al.* (2016, 2018, 2021)				x	x	
Liu *et al.* (2018)		x		x	x	
Maitra *et al.* (2018)				x	x	
Reyes *et al.* (2018)				x	x	
Story *et al.* (2018, 2020)		x	x			
Okoyomon *et al.* (2019)				x	x	
Emami *et al.* (2020, 2021)	x	x				
Peddinti *et al.* (2020)				x	x	
Jain *et al.* (2021)	x				x	
Gupta *et al.* (2021)	x				x	
Li *et al.*(2021)					x	
Wang *et al.*(2018)				x	x	
Gorla *et al.*(2014)				x		
Pandita *et al.* (2013)				x		
Qu *et al.* (2014)				x		
Gorla *et al.* (2014)				x		
Au *et al.* (2012)				x	x	
Petronella *et al.* (2014)				x	x	
Jiang *et al.* (2017)						x
Loyola *et al.* (2017)						x
Allamanis *et al.* (2016)						x
Alon *et al.* (2018)						x
Hu *et al.* (2018)						x
Sutskever *et al.* (2014)						x

Figure 1: Selection of closely-related, peer-reviewed publications in the past 10 years that generate notices. Column *'PN'* = ML/NLP-based; *'T'* = template-based; *'Q/A'* = question/answering based; *'In'* = inconsistency analysis; *'SA'* = static/dynamic analysis; and *'CS'* = code summarizing.

documentation. In this paper, we extend these efforts to the unique problem of detecting privacy behaviors of source code.

3 PRIVACY ACTION TAXONOMY (PACT)

As mentioned in Section 1, our goal is to detect and classify privacy behaviors of applications' source code. However, this is a challenging task for two reasons. First, we need to define the scope of relevant code segments in applications that access or use personal information. Second, we need to alleviate privacy subjectivity of annotators to identify privacy behaviors of these code segments. Each application's source code consists of several different files, which contain different classes and methods, and potentially any of these methods can use personal information to implement some functionality. Moreover, a single method can use personal information in multiple ways. For example, a method in a ride-sharing application can use location to find near-by rides and to share it with a third-party advertisement library. Consistently detecting all privacy behaviors in such methods becomes difficult due to the privacy subjectivity of annotators [44].

The complexity increases when we define a relevant code segment that is larger than a single method. To mitigate the complexity of consistently detecting the same privacy behaviors among multiple annotators, we need to create a taxonomy. Nickerson et al. [30] define a taxonomy for classifying objects as a set of dimensions (or categories) where each dimension has a set of characteristics (or labels) such that each object has one label under each category. We relax this definition for PAcT to include more than one label for each category. We provide the rationale for this modification with an example, later in this section.

Before explaining the details of PAcT, we describe the structure of the code segments we use to detect privacy behavior. In Android applications, the personal information (PI) can be accessed in two ways: System APIs[2] and User Interface (UI) [5, 14, 29]. In this paper, we focus on the usage of System APIs and leave the process of accessing the PI through UI for future. To access PI with system APIs, developers are required to declare necessary permissions in `AndroidManifest.xml`. In [15], we defined the code segments that call system APIs as *Permission-Requiring* Code Segments (*PRCS*). A PRCS may consist of multiple methods that are linked via call graph. This is because a method that calls an Android's *permission-requiring* system API to access personal information can use and further share the information with other methods. Since the PI can "hop" between methods in a PRCS, each method is referred to as a hop. In a PRCS, the "first hop" refers to the method that calls the *permission-requiring* API and each subsequent N^{th} hop refers to the method that is called by the $(N-1)^{th}$ hop. Each PRCS has at least one hop, i.e. the first hop. We use *PDroid* [15] to extract PRCS for 109 *permission-requiring* APIs (an additional 40 APIs than in [15]) from 12 different permission groups. We extract up to 3 hops for each PRCS (an additional 2 hops in comparison to [15]). We stop at 3 hops instead of tracing the complete flow of information from source to sink for the following reasons: (1) research suggests that 2-3 hops from a call graph provide sufficient information and anything beyond the 3 hops either becomes superfluous or challenging to handle [21]; (2) in our preliminary analysis, we found that in ~80% of the cases, the information returned by *permission-requiring* APIs is used within the first three hops; and (3) prior research has shown that the model's F-1 score decreases when we increase the number of lines of code [4, 15].

To develop PAcT, we follow the guidelines suggested by Nickerson et al. [30]. As mentioned in Section 1, we attempt to answer two questions regarding privacy actions of each PRCS: *how does a PRCS use personal information?* and *why?*. The answers to these two questions correspond to two categories of *Practice* and *Purpose* in our taxonomy, respectively. For each category, we create an initial set of labels: for *Practice* category, we follow the terminologies defined by GDPR [7] and CCPA [9]; whereas for *Purpose*, we use Apple's Privacy Nutrition Labels[3] as an initial guideline. Other works [6, 44] also introduce data practice labels. However, Wilson et al. [44] focus on web privacy policies pre-GDPR and CCPA, and Kumar et al. [6] focus only on Opt-in/Opt-out choices in policies.

The initial set of labels for *Practice* category include: `Processing`, `Collecting`, and `Sharing`. We formally define them as follows:

- **Processing**: When code uses a sensitive information for functionalities related to a first-party or a third-party library. It may include several operations - not limited to – retrieval, consultation, use, adaptation, or alteration.
- **Sharing**: When a first-party code shares a sensitive information with third-party code or if a third-party method calls a *permission-requiring* API.

[2]https://developer.android.com/reference/
[3]https://developer.apple.com/app-store/app-privacy-details/

- **Collecting**: When sensitive information is either explicitly saved in a persistent location off-device or implicitly saved in JSON objects or Maps, or returned beyond the last hop that indicates further use of information without calling *permission-requiring* API again.

The initial set of *Purpose* labels are `Functionality`, `Advertisement`, and `Analytics`. We formally define each label as follows:

- **Functionality**: When code executes a core functionality either in first-party or third-party library. (E.g. a messaging application accessing SMS messages).
- **Advertisement**: When code accesses PI for advertisement purposes, or an advertisement library accesses PI.
- **Analytics**: When code accesses PI for analyzing application's behaviors, or for analyzing/tracking user's behaviors.

We then expand these initial sets of labels using a conceptual-to-empirical approach [30]. In this step, we annotate PRCS over multiple iterations. In each iteration, the first two authors annotate each PRCS with the existing *Practice* and *Purpose* labels, and then analyze and discuss the annotations together to decide upon adding or removing labels from the existing label sets. If both agree that a label describes the privacy action of a PRCS, we keep that label and evaluate the ending conditions as shown in Appendix A - Figure A.2. If they do not agree or if ending conditions are not met, they conduct another round of iteration, else they finish the label expansion process. Figures A.4 and A.3, in Appendix A, show our final set of labels and their sub-labels for each category of PAcT. We describe how our initial set of *Practice* and *Purpose* labels evolved across iterations. In each iteration, we extract mutually exclusive PRCS.

– ***Iteration 1.*** We first randomly selected ~160 PRCS to annotate. We observed that several PRCS use third-party libraries to implement a core `Functionality`, such as authentication and payment services which provides additional information about the nature of `Functionality` label. Hence, we decided to add sub-labels when necessary. In iteration 1, we added two sub-labels, `Payment` and `Authentication`, under the `Functionality` label. We also discovered third-party libraries for measuring user-engagement and thus, added a sub-label, `User Experience`, under the `Analytics` label. Because of the additional sub-labels, we needed to go through another round of iteration.

– ***Iteration 2.*** We annotated a different set of randomly selected ~150 PRCS, and found two additional sub-labels, `Location` and `Marketing`, for `Functionality` and `Advertisement` respectively. We also identified `Crash Analytics` sub-label, for `Analytics` and created an `Other` label for *Practice* and *Purpose* categories to label one-off PRCS cases. Thus, we conducted another iteration.

– ***Iteration 3.*** We annotated another set of randomly selected ~140 PRCS, but we did not find any additional labels or sub-labels to add. Hence, we ended our iterative process and concluded that our taxonomy is complete.

As mentioned above, we omit a requirement proposed by Nickerson et al. [30] where each object must have one label under each category. This is because privacy practices are interrelated and it may be inaccurate to assign a single characteristic to each dimension. For example, if an application collects user id and email, it may use both to authenticate the user and to analyze the user's experience via a third-party library. Therefore, *Practices* for this application are `Processing` and `Sharing`, and *Purposes* are `Functionality` and `Analytics`. If we choose to label the *Practice* as only `Processing` or the *Purpose* as only `Functionality`, then privacy action classification is incomplete. Thus, each PRCS can have multiple labels which results in a multi-label multi-class dataset.

4 DATASET PREPARATION

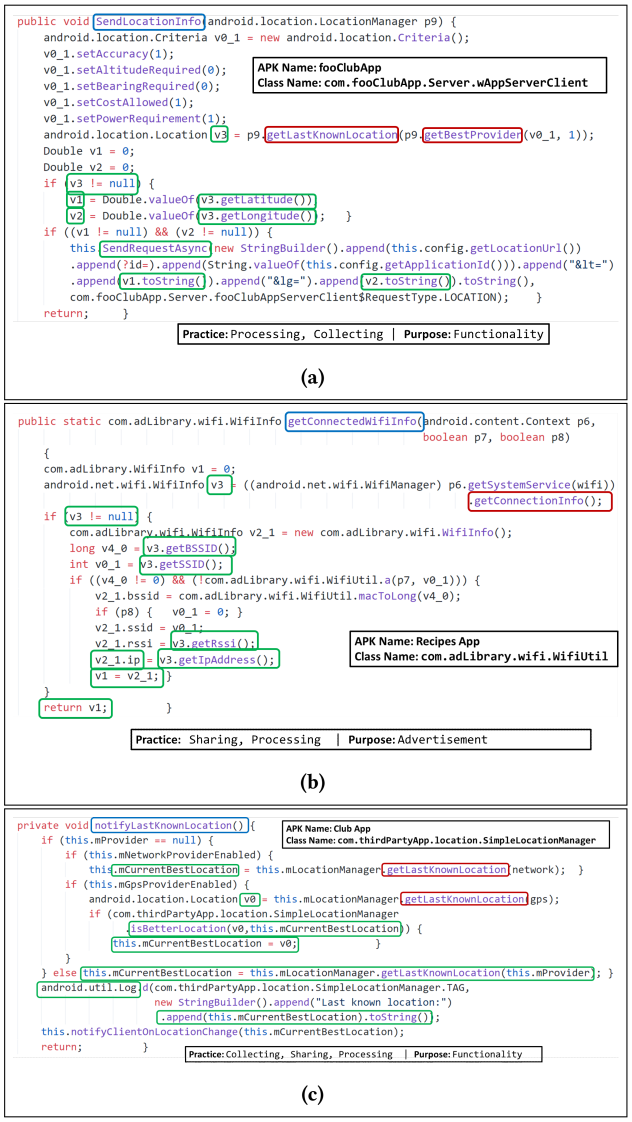
In this section, we discuss the process of preparing a dataset of annotated PRCS with their privacy actions' labels (defined in Section 3) to use for training and classification tasks. Since currently such publicly available dataset does not exist, we manually annotated and created the Annotated Dataset of Privacy Actions (ADPAc).

We downloaded 15,000 APK files from the curated PlayDrone dataset as part of Androzoo Collection [3] and extracted ~180,000 PRCS using PDroid [15]. From this set, we randomly selected ~5,200 PRCS to create ADPAc and another 100 PRCS to compute inter-annotator agreement between the two expert annotators (i.e. the first two authors). Note that, these PRCS are in addition to the ~450 PRCS we extracted to identify the labels of PAcT (i.e. Section 3). We used 2 expert annotators to balance the trade-off between a larger number of annotators and poor label quality. Research suggests that expert annotators are better than a larger number of annotators for tasks where subject expertise is required [31]. We computed the inter-annotator agreement to alleviate the subjectivity of the annotations to the extent possible. To help the annotation process, we also extracted some metadata about each code segment and provided them as comments in source code. The metadata includes the name of the APK file from which the PRCS is extracted, the class name, the method name, the list of *permission-requiring* APIs called in the first hop, and their descriptions. Finally, we developed a set of guidelines and an annotation tool *Codr*[4] to help annotators with labeling PRCS.

Annotation Guidelines. In addition to the definitions of the labels in PAcT, we developed a set of instructions to guide the annotators in detecting and annotating *Practice* and *Purpose* labels in PRCS. To help better understand these guidelines, we will explain a few annotated PRCS examples shared in Figure 2. In each sub-figure, we show a different PRCS sample with its labels and highlight aspects that annotators require for annotation. We highlight the method name in blue, *permission-requiring* APIs in red, and important variables and objects in green boxes. We also provide APK and class names in black boxes. We anonymize the APK, class, and certain object names to not draw attention to individual APKs or third-party libraries. The annotation instructions are as follows:
1. In each hop, analyze the following information:

- **Name of the method:** In most cases, the method name describes its behaviors which makes it easier to identify its *Practice* label. For example, `sendLocationInfo` in Figure 2(a) suggests that the method is *sending location information* [off-device] which indicates `Collecting` *Practice.*
- **Name of the APK and class:** Class and APK names help identify `Sharing` *Practice.* Class name of an application is derived from its package name. If this class name matches the APK name then it indicates that the code segment belongs to the APK file and is first-party code; otherwise, the code

[4]https://github.com/PERC-Lab/Codr



```
public void SendLocationInfo(android.location.LocationManager p9) {
    android.location.Criteria v0_1 = new android.location.Criteria();
    v0_1.setAccuracy(1);
    v0_1.setAltitudeRequired(0);
    v0_1.setBearingRequired(0);
    v0_1.setCostAllowed(1);
    v0_1.setPowerRequirement(1);
    android.location.Location v3 = p9.getLastKnownLocation(p9.getBestProvider(v0_1, 1));
    Double v1 = 0;
    Double v2 = 0;
    if (v3 != null) {
        v1 = Double.valueOf(v3.getLatitude());
        v2 = Double.valueOf(v3.getLongitude());    }
    if ((v1 != null) && (v2 != null)) {
        this.SendRequestAsync(new StringBuilder().append(this.config.getLocationUrl())
        .append(?id=).append(String.valueOf(this.config.getApplicationId())).append("&lt=")
        .append(v1.toString()).append("&lg=").append(v2.toString()).toString(),
        com.fooClubApp.Server.fooClubAppServerClient$RequestType.LOCATION);    }
    return;     }
```

APK Name: fooClubApp
Class Name: com.fooClubApp.Server.wAppServerClient

Practice: Processing, Collecting | **Purpose:** Functionality

(a)

```
public static com.adLibrary.wifi.WifiInfo getConnectedWifiInfo(android.content.Context p6,
                                                                boolean p7, boolean p8)
    {
    com.adLibrary.WifiInfo v1 = 0;
    android.net.wifi.WifiInfo v3 = ((android.net.wifi.WifiManager) p6.getSystemService(wifi))
                                                                        .getConnectionInfo();
    if (v3 != null) {
        com.adLibrary.wifi.WifiInfo v2_1 = new com.adLibrary.wifi.WifiInfo();
        long v4_0 = v3.getBSSID();
        int v0_1 = v3.getSSID();
        if ((v4_0 != 0) && (!com.adLibrary.wifi.WifiUtil.a(p7, v0_1))) {
            v2_1.bssid = com.adLibrary.wifi.WifiUtil.macToLong(v4_0);
            if (p8) {   v0_1 = 0; }
            v2_1.ssid = v0_1;
            v2_1.rssi = v3.getRssi();
            v2_1.ip = v3.getIpAddress();
            v1 = v2_1; }
    }
    return v1;          }
```

APK Name: Recipes App
Class Name: com.adLibrary.wifi.WifiUtil

Practice: Sharing, Processing | **Purpose:** Advertisement

(b)

```
private void notifyLastKnownLocation(){
    if (this.mProvider == null) {
        if (this.mNetworkProviderEnabled) {
            this.mCurrentBestLocation = this.mLocationManager.getLastKnownLocation(network);  }
        if (this.mGpsProviderEnabled) {
            android.location.Location v0 = this.mLocationManager.getLastKnownLocation(gps);
            if (com.thirdPartyApp.location.SimpleLocationManager
                   .isBetterLocation(v0,this.mCurrentBestLocation)) {
                this.mCurrentBestLocation = v0;              }
        }
    } else this.mCurrentBestLocation = this.mLocationManager.getLastKnownLocation(this.mProvider); }
    android.util.Log.d(com.thirdPartyApp.location.SimpleLocationManager.TAG,
                       new StringBuilder().append("Last known location:")
                       .append(this.mCurrentBestLocation).toString());
    this.notifyClientOnLocationChange(this.mCurrentBestLocation);
    return;         }
```

APK Name: Club App
Class Name: com.thirdPartyApp.location.SimpleLocationManager

Practice: Collecting, Sharing, Processing | **Purpose:** Functionality

(c)

Figure 2: Example PRCS with Metadata and their *Practice* and *Purpose* Labels to Demonstrate the Usage of Guidelines.

segment is third-party code. Information accessed by third-party code suggests `Sharing`. For example, if the APK name of an application is *"MyApp"* and the class name of a PRCS is `com.DemoLibrary.DemoClass`, it indicates that `DemoClass` class is not part of *"MyApp"* application, and thus it is third-party code (similar to Figure 2(b) which we label as `Sharing`). On the contrary, in Figure 2(a), the APK and class name match and indicate first-party code. Therefore, it is not a `Sharing` practice.

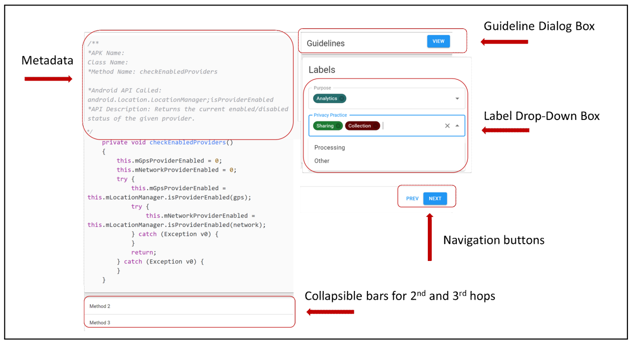
- ***Permission-Requiring* API(s) called:** Each PRCS calls these APIs to access or use sensitive information, which serve as good starting points inside a method to detect its *Practice*. In Figure 2(b), `getConnectionInfo` API returns Wifi information. The method then extracts IP and MAC addresses and stores them in `WifiInfo` objects which are returned by the method. This also indicates `Sharing` *Practice* labels, since the method returns information beyond the current hop (as defined in Section 3).
- **Android Developer Documentation:** This documentation[5] describes API usage of all system APIs in Android. In Figure 2(b) the method calls `getSystemService` API before calling `getConnectionInfo`. The documentation explains that when `getSystemService`[6] is called with "wifi" parameter, it returns `WifiManager` which is then used to *get connection info*. This corresponds to `Processing`.
- **Names and types of variables:** Variable names often describe purpose, especially variable(s) that store values returned by *permission-requiring* API(s). Analyzing how values stored in them are accessed or changed helps detecting *Practice* labels. In Figure 2(c), variable `mCurrentBestLocation` stores the location information returned by `getLastKnownLocation`. The name distinctly indicates that it is used to store *the current best location*. If variable names are obfuscated, we analyze the variable type to infer its purpose. In Figure 2(a), the variable `v3` is a `Location` object, thus, it must be used to store location information.

2. Analyze the above information for each PRCS to understand how a PRCS uses personal information. Then, trace the flow of information between hops to check if they return sensitive information directly, after processing it, or do not return it at all. Based on the definitions, identify its *Practice*. For example, in Figure 2(b), the method returns `WifiInfo`. If the PRCS has a second or third hop, we trace how Wifi information is further used to identify the labels.
3. If the class name does not match with the APK name, then search for the class name on the web to identify if the third-party code belongs to an advertisement or analytics library. For example, in Figure 2(b) the name of the APK file and class name do not match. Searching for "`adLibrary`" class on the web indicates that it is an advertisement library. Therefore, the purpose here is `Advertisement`.
4. In case *Practice* or *Purpose* of a method does not match with definitions of any label and is unique, annotate that sample as `Other`.

Annotating PRCS using Codr: We developed a web-based annotation tool called *Codr* to annotate PRCS with *Practice* and *Purpose*. There are several annotation tools available for various annotation tasks [18, 41]. However, none of them allow source code formatting, which is a fundamental requirement to create ADPAc, since without any formatting annotators would find it difficult to read and comprehend the source code. Codr also supports other features such as: (i) having multiple annotators for each annotation project, (ii) creating individual annotator roles and granting permissions, (iii) being hosted on a private server to ensure better privacy, and (iv) having features such as classification, highlighting, and annotation of text either in source code or in natural language.

Figure 3 shows the interface of *Codr*. In *Codr*, an annotator can view all the three hops of a PRCS and the guidelines for the annotation task. The tool also includes two drop-down menus that consist of sub-labels of *Purpose* and *Practice*. *Codr* has a text field, where annotators can suggest additional labels as well. We used *Codr* to

[5]https://developer.android.com/docs
[6]https://developer.android.com/reference/android/content/Context#WIFI_SERVICE



```
/**
*APK Name:
Class Name:
*Method Name: checkEnabledProviders

*Android API Called:
android.location.LocationManager;isProviderEnabled
*API Description: Returns the current enabled/disabled
status of the given provider.
*/
    private void checkEnabledProviders()
    {
        this.mGpsProviderEnabled = 0;
        this.mNetworkProviderEnabled = 0;
        try {
            this.mGpsProviderEnabled =
this.mLocationManager.isProviderEnabled(gps);
            try {
                this.mNetworkProviderEnabled =
this.mLocationManager.isProviderEnabled(network);
            } catch (Exception v0) {
            }
            return;
        } catch (Exception v0) {
        }
    }
```

Figure 3: Screenshot of *Codr*, our Annotation Tool.

annotate ~5,200 PRCS. Since, each PRCS can have multiple *Practice* and *Purpose* labels, we ended up with a multi-label multi-class dataset. Table 1 shows the distribution and percentage of samples annotated for each label. For example, there are 3,407 PRCS (~66%) in the ADPAc annotated with `Processing` and 28% with `Other` in *Practice* category. This number is the sum of all segments annotated using the `Other` label with (807 PRCS (16%)) or without (632 PRCS (12%)) other types of *Practices*. Similarly, the counts for segments annotated using the `Other` label with and without other *Purposes* are 378 (7%) and 273 (5%), respectively. This shows that we can identify 88% and 95% of PRCS with *Practice* and *Purpose* labels, with at least one label different from the `Other` label. However, using the "`Other`" label in privacy notices may result in vagueness. To resolve this, we plan to evaluate the code segments labeled as `Other` with respect to regulations and best practices to identify new labels that could better describe the privacy behavior.

Table 1: Distribution of Labels in our Annotated Dataset.

Label	Count	Label	Count
`Processing`	3,407 (66%)	`Functionality`	3,606 (69%)
`Sharing`	2,425 (47%)	`Analytics`	1,204 (23%)
`Collecting`	970 (19%)	`Advertisement`	979 (19%)
`Other`(*Practice*)	1,439 (28%)	`Other` (*Purpose*)	651 (13%)

Inter-Annotator Agreement: To ensure that the annotators label PRCS reliably, we compute inter-annotator agreement scores using Fleiss' Kappa[7]. The annotators labeled the randomly selected 100 PRCS with *Practice* and *Purpose*. As mentioned in Section 3, a PRCS may contain more than one label for *Practice* or *Purpose*. For instance, $Annotator_1$ may label a PRCS as `Sharing` and `Processing`, whereas $Annotator_2$ may label the same PRCS as only `Sharing`. Hence, we consider a *range of agreement*. For the lower bound, if the two annotators do not annotate a PRCS with the exact same labels, we consider this as a disagreement. For instance, we regarded the annotations in the previous example as a disagreement. To compute the upper bound, we consider the two annotations as an agreement, if there is at least one common label. Therefore, in the previous example, the two annotations are regarded as an agreement since they have a common label, `Sharing`. The range of agreement for *Practice* label is 0.20 - 0.70. The 0.7 value for Kappa entails a substantial agreement between the annotators for at least one common *Practice* label. The lower bound score of 0.2 entails a "slight" agreement. We manually inspected annotations to better understand this score range, and found that there is a complete agreement between the annotators in 29 cases and a disagreement in 71 cases out of 100 PRCS annotations. However, in 54 out of the 71 disagreement cases, annotations differed only by a single label. For example, for a particular PRCS, $Annotator_1$ labeled it as `Sharing`, `Collecting`, and `Processing` whereas $Annotator_2$ labeled it as `Sharing` and `Collecting`. This implies that the annotators either had a complete agreement or a disagreement for a single label in 83 out of 100 annotations which indicates a high agreement. Fleiss' kappa for *Purpose* label is 0.65 which means a substantial agreement between the annotators for *Purpose* label. We achieved the same score for both lower and upper bounds for *Purpose* labels. This agreement metrics demonstrate the efficacy of creating a dataset of PRCS and their privacy behavior with multiple annotators.

[7]https://www.statisticshowto.com/fleiss-kappa/

5 EXPERIMENT DESIGN

In this section, we describe the experiments we conducted with the ADPAc dataset to detect *Practices* and *Purposes* of PRCS.

5.1 Research Questions

Our research questions, in this paper, are as follows:

RQ1. *What is the baseline accuracy to identify individual Practice and Purpose labels?* Our goal is to establish performance baselines in detecting these labels from PRCS, using standard ML performance metrics which serves as a base for evaluating future improvements.

RQ2. *Does increasing the number of hops impact the performance of models?* In this work, a PRCS may include up to 3 hops and each hop provides additional information about processing of personal information, which may or may not be useful to identify *Practice* or *Purpose*. We assess how each additional hop impacts the model's performance for each label to help understand if we need to extract the complete information flow.

5.2 Methodology

To conduct the experiments, we will (1) prepare datasets for each research question, (2) create an embedding model to use in each experiment, (3) create AST path representations of PRCS, and (4) develop and train a simple Neural Network model.

Generating Training Data: As mentioned in Section 4, our dataset is a multi-label multi-class dataset (ADPAc), i.e. each PRCS is annotated with multiple labels of *Practice* and *Purpose*. However, in this paper, we formulate the label identification as a binary classification task i.e., a separate model to identify each of the *Practice* or *Purpose* labels. This is to analyze and determine if different labels require different configurations (such as number of hops) for accurate classification. This analysis provides us with insights about how each data practice is implemented in source code and across different hops. Furthermore, a binary classification task facilitates creating a performance baseline that needs to be met when creating a unified multi-label multi-class model.

We, first, converted ADPAc to 8 individual binary class datasets corresponding to the 8 labels of *Practice* and *Purpose*. In each dataset, we ensure that there is an equal number of positive and negative samples. Positive samples are the PRCS that are annotated with one label, for example, `Collecting` in *Practice* category, and negative samples are PRCS annotated with all the other labels except `Collecting`. In case a code segment contains two PAcT labels, we created two copies of the sample and ensured that the same sample is not repeated in both the positive and negative entries in the respective binary datasets. To address **RQ1**, for each sample in each binary dataset, we only extracted the code of the *first* hop, even if the sample contained more than one hop. For **RQ2**, we first filtered the 8 binary datasets to only include samples that contain 3 hops and then balanced the datasets. Next, we created three derived datasets for each label from the label's binary dataset, where each of the three derived datasets contained the exact same samples but varied in the number of hops. For example, we created 3 datasets for `Sharing` – $Sharing_{1Hop}$, $Sharing_{2Hop}$, and $Sharing_{3Hop}$. All three contain the same samples, with $Sharing_{1Hop}$ containing only the first hop; $Sharing_{2Hop}$ containing the first and second hops; and, $Sharing_{3Hop}$ containing all the 3 hops. Since, there are 8 labels, we ended up with 24 datasets in total (3 datasets for each of the 8 labels). Note that, the dataset of RQ2 with only 1 hop is different than the dataset of RQ1 – while RQ1 contains the 1st hop information from all samples, RQ2 contains only the 1st hop information for samples with 3 hops.

After creating RQ1 and RQ2 datasets, we used a tool called `astminer` [19] to extract the Abstract Syntax Tree (AST) paths from the source code. We used AST paths instead of source code as input, since the paths contain more structural information about the code than the source code text. AST paths are also the standard in code summarization work [4, 20]. To understand more about AST paths, please refer to the Appendix Section A.3.

Creating Embedding: Next, we created an embedding model to generate token representations. For this, we downloaded another ∼10,000 APK files from Androzoo collection [3], extracted ∼100,000 PRCS using *PDroid* [15] (different from the ∼5,200 PRCS we annotated earlier), and used `astminer` to extract the AST paths of all the code samples. We then used Gensim[8] to create a skip-gram embedding model of the extracted AST paths. For this model, window size and embedding dimension are the hyperparameters, and we chose them as 3 and 100 respectively based on the results of our pilot studies.

AST Path Representations: As described in Appendix Section A.3, each AST path contains 3 tokens where the second token is a sequence of non-terminal nodes. We do not split the second token into individual non-terminal nodes, but instead consider it as a single token – since the influence of individual non-terminal nodes in comparison to the sequence is uncertain. In each PRCS P, each AST path p, is represented as $p = [t_s; t_n; t_e]$ where t_s and t_e are *start* and *end* terminal nodes and t_n is a sequence of non-terminal nodes. Each sample P consists of several paths p_i depending on the length of the source code. Since a model requires a fixed input length, we only select *num_paths* number of paths for each sample. If a sample contains more than *num_paths* paths, we consider the first set of paths and remove the rest. On the other hand, we pad the sample with zeros if there are fewer than *num_paths* paths. Therefore $P = \{p_1, p_2, \ldots, p_{num\_paths}\}$ where each p_i is an individual path.

[8]https://radimrehurek.com/gensim/

Model Architecture: We base our binary classification model on Code2Seq [4] but we modify it in two ways: (i) we do not use an attention layer. For translation tasks, an attention layer is added to the encoder-decoder models to identify relevant tokens in input and help decoder generate output. For a binary classification task, an attention layer increases the model's complexity since it increases the number of parameters to train, even for a small dataset such as ours. Hence, we omit the attention layer. (ii) we replace the decoder with fully-connected layers, since we perform a classification task. Decoder generates sequences from hidden states provided by the encoder. In our work, we do not need to generate any sequences.

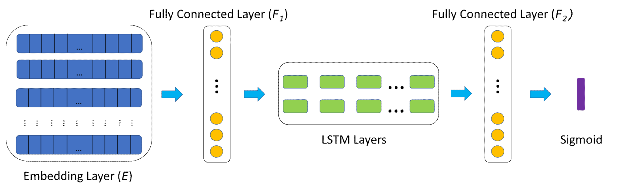
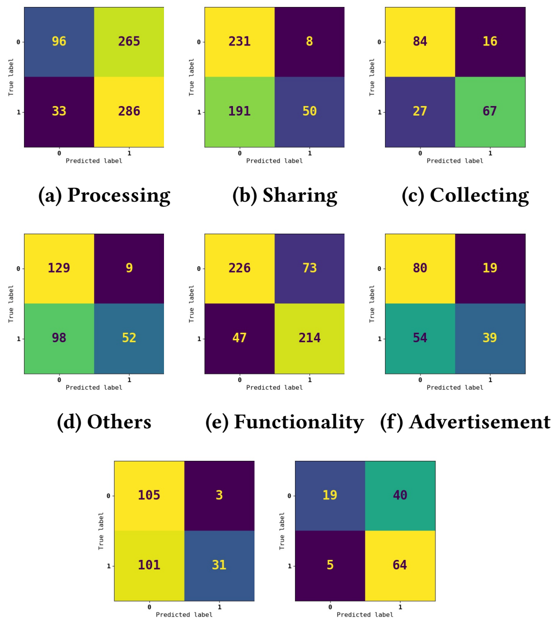


Figure 4: The Architecture of our RNN Model.

Figure 4 shows our model architecture. The first part is the embedding layer, where we load our trained embedding model E and extract embedding for each token in an AST path. After embedding, each AST path $[t_s; t_n; t_e]$ becomes $embedded\_path = [E_{t_s}; E_{t_n}; E_{t_e}]$. For an embedding dimension of *embed_dim*, each *embedded_path* becomes a $3 \times embed\_dim$ matrix. There are *num_paths* paths in a PRCS sample P, therefore, each P is $num\_paths \times 3 \times embed\_dim$ tensor. To represent each *embedded_path*, we concatenate the tokens to form a single vector of size $3 \times embed\_dim$ and then apply a fully-connected layer to reduce the dimension to *embed_dim*,since in our earlier experiments, we found that using a smaller representation for a path is very effective given our dataset size and the model architecture. Thus, the output of the fully-connected layer is a $num\_paths \times embed\_dim$ matrix. We then encode this output with 2 LSTM layers and apply a recurrent dropout of 0.5. We use the final hidden state of LSTM layers as an input to a fully connected layer with 100 units and then apply a Sigmoid non-linearity. We conducted preliminary experiments and found the following model hyperparameters are the most optimal for RQ1 and RQ2 tasks: we use binary cross-entropy loss and Adam optimizer [17] with a learning rate of 1e-5 for 50 epochs; chosen batch size is 8 and clip size is 5; number of AST paths is 100; and as mentioned earlier, embedding dimension is 100.

For both RQ1 and RQ2, we use 80:10:10 split ratio of samples for training, validation, and test set, respectively. After 50 epochs, we load the model with the highest validation accuracy and report the model's accuracy and F1 scores on the test set. We also show the test set confusion matrices to draw additional insights. The only difference between the experimental setup of RQ1 and RQ2 is that in RQ1, we select *the first* 100 paths from a sample while in RQ2, we *randomly select* these 100 paths. This is because when we extract AST paths from a sample that contains three hops, the extracted paths include first hop paths, followed by the second and third hop paths. Since in RQ2, we aim to analyze how each additional hop impacts the models' performance, we randomly select the paths to ensure having enough representation from all the three hops. We



(a) Processing (b) Sharing (c) Collecting

(d) Others (e) Functionality (f) Advertisement

(g) Analytics (h) Others

Figure 5: RQ1 Confusion Matrices. *Practice*: Sub-figures (a), (b), (c), (d); *Purpose*: Sub-figures (e), (f), (g), (h). 0 is positive label and 1 is negative label in each dataset.

use PyTorch 1.8.1 in Python 3.7 to train and test the models on a workstation with a Xeon CPU, 54 GB RAM, and a Tesla T4 GPU.

6 EXPERIMENT RESULTS

Tables 2 and 3 show the scores for RQ1 and Figure 6 shows the scores for RQ2 experiments. We also analyze confusion matrices for a better interpretability of models' performances (i.e. Figure 5 and Figures 7 - 14 for RQ1 and RQ2 confusion matrices, respectively). In each confusion matrix, 0 denotes the positive label and 1 denotes the negative label of the dataset and the horizontal axis is for the predicted label and the vertical is for the true label.

6.1 RQ1: What is the baseline accuracy to identify each *Practice* and *Purpose* labels?

Remember from Section 5.2 that the data for RQ1 contains 8 balanced datasets, with one hop for each sample. Out of all *Practice* labels, our model identifies `Collecting` with the highest accuracy of 77.84% and F-1 score of 79.62%. The confusion matrix for `Collecting` shows larger counts for true positives and true negatives resulting in a true positive rate of 0.84 and a true negative rate of 0.71 which demonstrates that the model is not biased towards either the positive or the negative label. While we do not weigh false positives and false negatives differently in our current model, we prefer to have higher recall than precision because we want to maximize the coverage of detecting a particular privacy action. Since a developer will consume the results of our model to write privacy notices, having a false positive is not as harmful compared to missing out detecting a particular behavior. This is because, in case the model predicts a false positive label, say `Collecting`, developers can verify the prediction and not include the *Practice* in their notice. On the other hand, if the model does not detect `Collecting` *Practice* when it is present, developers will not create such a notice making it inconsistent with the application's privacy behavior. Our model achieves the next best scores for identifying `Sharing` and `Other` *Practices* with F-1 scores of 69.89% and 70.68%, respectively. For both of these *Practices*, we observe a similar pattern in the confusion matrices – a high recall value and low precision. For instance, for `Sharing`, we see a precision of 55% and a recall of 97%. This shows that the model is not able to identify `Sharing` or `Other` with only one hop and requires multiple hops to classify them (as discussed in the next sub-section).

Table 2: Metric Scores for Identifying *Practices* in PRCS.

Positive Label	Accuracy	F-1 Score
`Processing`	56.18%	39.18%
`Sharing`	58.54%	69.89%
`Collecting`	**77.84%**	**79.62%**
`Other` (*Practice*)	62.85%	70.68%

Table 3: Metric Scores for Identifying *Purposes* in PRCS.

Positive Label	Accuracy	F-1 Score
`Functionality`	**78.57%**	**79.02%**
`Advertisement`	61.98%	68.67%
`Analytics`	56.67%	66.88%
`Other` (*Purpose*)	64.84%	45.78%

Our model achieves the lowest scores for detecting `Processing` *Practice*. Based on the confusion matrix in Figure 5(a), our model predicts the negative label more often. This is in contrast with all the other confusion matrices of *Practices* labels, where the true positive rates are higher, because `Processing` is a more complex *Practice* and encompasses different operations – such as, using location to calculate distance, checking the network connection type, or retrieving available accounts. Thus, *Practice* cannot be identified by the presence of an API call, an information type, or a single hop. On the other hand, it is easier to identify when a PRCS is **not** `Processing`, and as a result, the count for predicted negatives is higher. We also observe from the confusion matrices for all *Practices*, with the exception of `Processing`, that the count for false negative is much lower than the count for false positive. This demonstrates that our model is most likely to detect a *Practice*, correctly or incorrectly, than miss a *Practice* that is present. However, to improve our model's performance, we need to include more paths from the second or third hops (which we discuss in the next sub-section).

Our model identifies `Functionality` *Purpose* with the highest accuracy and F-1 scores of 78.57% and 79.02%. This is because code segments written for `Functionality` are often first-party code composed of one or two methods that access personal information. The sufficiency of one hop is also evident from the confusion matrix in Figure 5(e), where we observe a low count of false negatives and false positives. For `Advertisement` and `Analytics` *Purposes*, our model achieves low accuracy (61.98% and 56.67%) and F-1 (68.67% and 66.88%). The confusion matrices in Figure 5(f) and (g) show that the recall is high while precision is low, a pattern we observed for Sharing *Practice* as well. Sharing is generally correlated with Advertisement and Analytics *Purposes*, since they all involve third-party libraries (as defined in Section 3). Both labels have low false negative counts which demonstrate our model's efficacy.

Lastly, for Other *Purpose*, our model achieves an accuracy and F-1 scores of 64.84% and 45.78%. Detecting Other is also a difficult task since it does not have a specific feature to help identify them. On the contrary, it is easier to identify **not** Other labels, which consist of Functionality, Advertisement, and Analytics labels. Thus, as shown in Figure 5(h), the model predicts negative label more often than positive label resulting in more false negatives. We suggest that increasing the number of hops could help improve the performance. Overall, we observe that our model's performance differs based on the types of *Practice* and *Purpose* labels.

6.2 RQ2: Does increasing the number of hops impact the performance of the model?

As described in Section 5.2, data for RQ2 contains 24 balanced datasets, with 3 datasets containing different number of hops for each label. Before evaluating the impact of increasing the number of hops, we compare the effect of using randomized paths. For example, if a model can detect a label from five AST paths out of 100 then randomly selecting these paths could adversely effect the model's accuracy since these relevant paths may not be selected each time. On the other hand, if the model can identify a label from 30 paths, then the model has a higher probability of correctly detecting the label each time. As described in Section 5, a key difference between the RQ1 and RQ2 experiments is that in RQ1, we select *the first* 100 paths while in RQ2, we *randomly select* 100 paths from each sample. When we compare the scores for 1 hop of RQ2 in Figure 6 with the scores from RQ1 in Tables 2 and 3, we observe noticeable differences between the two sets of metrics. In some cases, the performance of the model decreased while in others it increased. For example, our model's accuracy for classifying Collecting is ~78% in RQ1 whereas it is ~70% for 1 hop in RQ2. On the other hand, the accuracy for Functionality increased from 79% in RQ1 to 86% in RQ2. This inconsistency between the two quantitative results is related to random path selection as well as the proportion of relevant paths in a sample, which can affect the model's performance. A sample with a higher proportion of relevant paths is impacted differently (and probably less) due to random path selection compared to the one with fewer relevant paths. This inconsistency in the model's performance further demonstrates the complexity and challenges to detect privacy behavior in a code segment.

Figure 6(a),(b) and Figures 7 - 10 show the quantitative scores and confusion matrices for identifying *Practice* labels for RQ2. In summary, we notice the largest gain in accuracy by 6 points between 1 and 2 hops when classifying Processing and similar gain between 1 and 3 hops for Sharing. We also observe a large gain of almost 9 points in F-1 score between 1 and 2 hops for Processing. The F-1 scores also improve by 8 and 14 points, for Sharing and Other.

As shown in Figure 6, although the performance of our model in detecting Processing label increases when adding a second hop, any additional hop (i.e. adding the third hop) does not impact the performance of the model. Similarly, the confusion matrices in Figure 7 do not show any noticeable difference between the second and third hops. The main reason is that most often code segments Process personal information within the first two hops. Therefore, adding a third hop does not affect the performance. Improvement beyond the 64.75% accuracy is hard due to the complexity of Processing label as discussed in RQ1.

Labels	1 Hop	2 Hops	3 Hops
Processing	58.75%	**64.75**%	64.25%
Sharing	65.1%	67.1%	**71.09**%
Collecting	70.39%	68.42%	**71.09**%
Other	66.5%	**69.91**%	69%

(a) Accuracy Scores (Practice)

Labels	1 Hop	2 Hops	3 Hops
Processing	51.04%	**59.83**%	59.49%
Sharing	69.82%	73.39%	**77.58**%
Collecting	**74.58**%	73.03%	74.36%
Other	59.06%	68.87%	**72.99**%

(b) F-1 Scores (Practice)

Labels	1 Hop	2 Hops	3 Hops
Functionality	85.9%	**86.97**%	85.64%
Advertisement	70.24%	78.57%	**79.76**%
Analytics	61.98%	**70.83**%	**70.83**%
Other	**72.5**%	68.8%	**72.5**%

(c) Accuracy Scores (Purpose)

Labels	1 Hop	2 Hops	3 Hops
Functionality	**88.89**%	87.59%	86.43%
Advertisement	73.96%	80%	**80.46**%
Analytics	70.45%	**75.65**%	74.07%
Other	**73.81**%	69.14%	73.17%

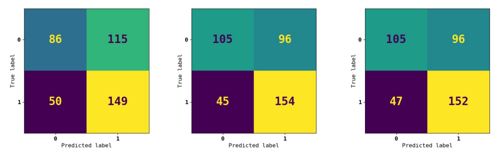
(d) F-1 Scores (Purpose)

Figure 6: RQ2 Metric Scores. Top: *Practice* Labels. Bottom: *Purpose* Labels. Bold values indicate the highest score in each row.

Our model's performance increases with every hop while detecting Sharing label. Adding the second hop increases the accuracy by 2 points and F-1 score by 3 points, while including the third hop improves the accuracy and F-1 score by another 4 points each. As stated in RQ1, Sharing often occurs in the second or third hop, which is the reason that the model's performance increases with more hops. The confusion matrices in Figure 8 illustrate the increase in the count of true positive predictions from 155 in Figure 8(a) to 192 in (c). The count of false negative decreases with each subsequent hop, as well.

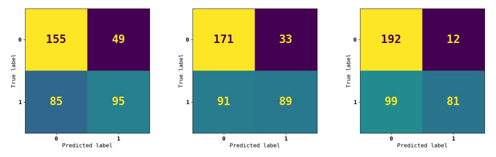
The model's accuracy and F-1 score improves for Other *Practice* when we add more hops. We notice a consistent performance for Collecting, with a small dip when we add the second hop information. This is because the model could be over indexing on the personal information being returned by the first hop, without analyzing how it is used in the second or third hops. The confusion matrices in Figure 9 also show that adding each additional hop marginally helps the model identify **not** Collecting *Practices*. Overall, adding multiple hops does not impact the performance for this task. It impacts the count for false negative with additional hops but the increase in the count is small.

Figure 6(c), (d) and Figures 11 - 14 show the quantitative scores and confusions matrices for identifying *Purposes* labels. We observe overall increases in performance scores for Advertisement and Analytics. The accuracy and F-1 scores increase with each additional hop for both labels. Their confusion matrices in Figures 12 and 13 illustrate that the count for true positives relatively stays



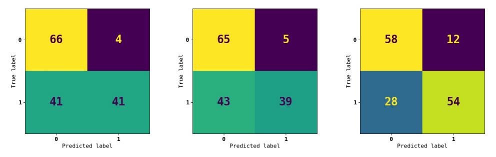
(a) 1 Hop (b) 2 Hops (c) 3 Hops

Figure 7: Hop Analysis- `Processing`



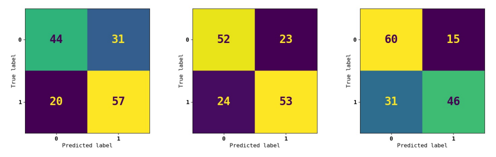
(a) 1 Hop (b) 2 Hops (c) 3 Hops

Figure 8: Hop Analysis- `Sharing`



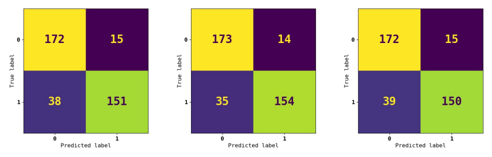
(a) 1 Hop (b) 2 Hops (c) 3 Hops

Figure 9: Hop Analysis - `Collecting`



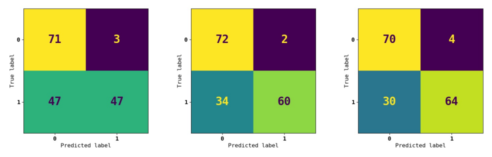
(a) 1 Hop (b) 2 Hops (c) 3 Hops

Figure 10: Hop Analysis - `Other`(*Practice*)



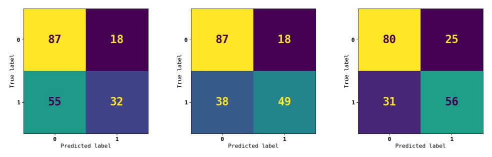
(a) 1 Hop (b) 2 Hops (c) 3 Hops

Figure 11: Hop Analysis - `Functionality`



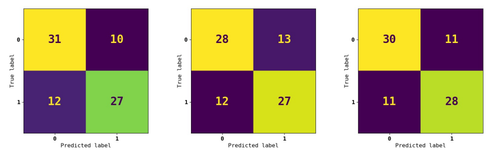
(a) 1 Hop (b) 2 Hops (c) 3 Hops

Figure 12: Hop Analysis - `Advertisement`



(a) 1 Hop (b) 2 Hops (c) 3 Hops

Figure 13: Hop Analysis - `Analytics`



(a) 1 Hop (b) 2 Hops (c) 3 Hops

Figure 14: Hop Analysis - `Other` (*Purpose*)

the same across all three hops, while the count for true negative increases. This is because the model is learning to differentiate between false negatives and true negatives, thereby improving the performance for both labels. We do not notice a significant difference in the performance when classifying `Functionality` since the first hop itself determines if an application uses personal information for `Functionality`. Adding additional hops includes paths from other hops and may add noise. The confusion matrices in Figure 11 also confirms that additional hops do not improve the performance. Similarly, the performance does not improve for `Other`, which indicates that the model does not require additional hops to identify this label, as well. The count for false negative is constant for almost all labels with each additional hops which is promising. Overall, when identifying *Purposes*, including more hops helps the model differentiate between true negative and false negative samples; thus, improves the performance for some labels.

7 CONCLUSION AND FUTURE WORK

In this paper, we developed an automated approach to help detect privacy behavior of Android code segments which access or use personal information. These labels can help developers modify relevant sections of their generated privacy policy or provide intermediate notices to legal experts writing privacy policies, thereby reducing the number of inconsistencies between applications' code and privacy notices. We defined privacy behaviors in terms of *Practice* and *Purpose* categories, and developed a Privacy Action Taxonomy (PAcT) to identify various labels within these categories. We manually created ADPAc, a dataset of ∼5,200 code segments annotated with *Practice* and *Purpose* labels and then developed a simple RNN model to conduct binary classifications in two sets of experiments: one to establish the baseline accuracy for classifying each individual label, and second to measure the impact of including multiple hops for the same task. We achieved the highest F-1 scores of 79.62% and 79.02% across the label types for *Practice* and *Purpose*, respectively. These results demonstrate the efficacy of our approach in detecting privacy behavior in source code.

We focused on binary classification in this paper to analyze if different labels require different configurations, and to gather insights about how each data practice is implemented in source code and across different hops. In future, we plan to include UI code segments in our analysis. We will also conduct multi-label multi-class classification experiments by developing more complex models, such as Convolutional Graph Neural Networks (ConvGNN) [21], and train them to identify all *Practice* and *Purpose* labels for each code segment. We will use attention layers to analyze which code segment features the models rely on to determine *Practice* or *Purpose* labels.

REFERENCES

[1] 2017. Privacy and Data Security Update (2016)). Federal Trade Comission. https://www.ftc.gov/reports/privacy-data-security-update-2016.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. *International Conference on Learning Representations* (2019).

[5] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. 2017. Uiref: analysis of sensitive user inputs in android applications. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 23–34.

[6] Vinayshekhar Bannihatti Kumar, Roger Iyengar, Namita Nisal, Yuanyuan Feng, Hana Habib, Peter Story, Sushain Cherivirala, Margaret Hagan, Lorrie Cranor, Shomir Wilson, et al. 2020. Finding a choice in a haystack: Automatic extraction of opt-out statements from privacy policy text. In *Proceedings of The Web Conference 2020*. 1943–1954.

[7] European Union. 2021 (accessed Oct 1st, 2021). The EU General Data Protection Regulation (GDPR). https://gdpr-info.eu/.

[8] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*. 1025–1035.

[9] Government of California. 2021 (accessed Oct 1st, 2021). California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa.

[10] Irit Hadar, Tomer Hasson, Oshrat Ayalon, Eran Toch, Michael Birnhack, Sofia Sherman, and Arod Balissa. 2018. Privacy by designers: software developers' privacy mindset. *Empirical Software Engineering* 23, 1 (2018), 259–289.

[11] Mitra Bokaei Hosseini, Travis D. Breaux, Rocky Slavin, Jianwei Niu, and Xiaoyin Wang. 2021. Analyzing privacy policies through syntax-driven semantic analysis of information types. *Information and Software Technology* 138 (2021), 106608. https://doi.org/10.1016/j.infsof.2021.106608

[12] Mitra Bokaei Hosseini, Xue Qin, Xiaoyin Wang, and Jianwei Niu. 2018. Extracting information types from android layout code using sequence to sequence learning. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.

[13] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[14] 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 *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 977–992.

[15] Vijayanta Jain, Sanonda Datta Gupta, Sepideh Ghanavati, and Sai Teja Peddinti. 2021. PriGen: Towards Automated Translation of Android Applications' Code to Privacy Captions. (2021).

[16] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 135–146.

[17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[18] Jan-Christoph Klie. 2018. INCEpTION: Interactive machine-assisted annotation.. In *DESIRES*. 105.

[19] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2019. PathMiner: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 13–17.

[20] Alexander LeClair, Zachary Eberhart, and Collin McMillan. 2018. Adapting neural text classification for improved software categorization. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 461–472.

[21] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *28th ACM/IEEE International Conference on Program Comprehension (ICPC'20)*.

[22] Tianshi Li, Elijah B Neundorfer, Yuvraj Agarwal, and Jason I Hong. 2021. Honeysuckle: Annotation-Guided Code Generation of In-App Privacy Notices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 5, 3 (2021), 1–27.

[23] Xueqing Liu, Yue Leng, Wei Yang, Wenyu Wang, Chengxiang Zhai, and Tao Xie. 2018. A large-scale empirical study on android runtime-permission rationale messages. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 137–146.

[24] Xueqing Liu, Yue Leng, Wei Yang, Chengxiang Zhai, and Tao Xie. 2018. Mining android app descriptions for permission requirements recommendation. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 147–158.

[25] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 287–292.

[26] Sayan Maitra, Bohyun Suh, and Sepideh Ghanavati. 2018. Privacy Consistency Analyzer for Android Applications. In *2018 IEEE 5th International Workshop on Evolving Security & Privacy Requirements Engineering (ESPRE)*. IEEE, 28–33.

[27] Abraham H Mhaidli, Yixin Zou, and Florian Schaub. 2019. " We can't live without them!" app developers' adoption of ad networks and their considerations of consumer risks. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*. 225–244.

[28] Hosseini Mitra Bokaei, SudarshanzWadkar, Breaux Travis D, and Jianwei Niu. 2016. Lexical similarity of information type hypernyms, meronyms and synonyms in privacy policies. In *2016 AAAI Fall Symposium Series (AAAI '16)*.

[29] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. Uipicker: User-input privacy identification in mobile applications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 993–1008.

[30] Robert C Nickerson, Upkar Varshney, and Jan Muntermann. 2013. A method for taxonomy development and its application in information systems. *European Journal of Information Systems* 22, 3 (2013), 336–359.

[31] Stefanie Nowak and Stefan Rüger. 2010. How reliable are annotations via crowdsourcing: a study about inter-annotator agreement for multi-label image annotation. In *Proceedings of the international conference on Multimedia information retrieval*. 557–566.

[32] Ehimare Okoyomon, Nikita Samarin, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, Irwin Reyes, Álvaro Feal, and Serge Egelman. 2019. On the ridiculousness of notice and consent: Contradictions in app privacy policies. (2019).

[33] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. {WHYPER}: Towards Automating Risk Assessment of Mobile Applications. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 527–542.

[34] Sai Teja Peddinti, Igor Bilogrevic, Nina Taft, Martin Pelikan, Úlfar Erlingsson, Pauline Anthonysamy, and Giles Hogben. 2019. Reducing Permission Requests in Mobile Apps. In *Proceedings of the Internet Measurement Conference*. 259–266.

[35] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1354–1365.

[36] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, and Serge Egelman. 2018. "Won't somebody think of the children?" examining COPPA compliance at scale. *Proceedings on Privacy Enhancing Technologies* 2018, 3 (2018), 63–83.

[37] Sanae Rosen, Zhiyun Qian, and Z Morely Mao. 2013. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and application security and privacy*. 221–232.

[38] Mark Rowan and Josh Dehlinger. 2014. Encouraging privacy by design concepts with privacy policy auto-generation in eclipse (page). In *Proceedings of the 2014 Workshop on Eclipse Technology eXchange*. 9–14.

[39] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. 2016. PVDetector: a detector of privacy-policy violations for Android apps. In *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*. IEEE, 299–300.

[40] 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*. 25–36.

[41] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. 2012. brat: a Web-based Tool for NLP-Assisted Text Annotation. In *Proceedings of the Demonstrations Session at EACL 2012*. Association for Computational Linguistics, Avignon, France.

[42] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

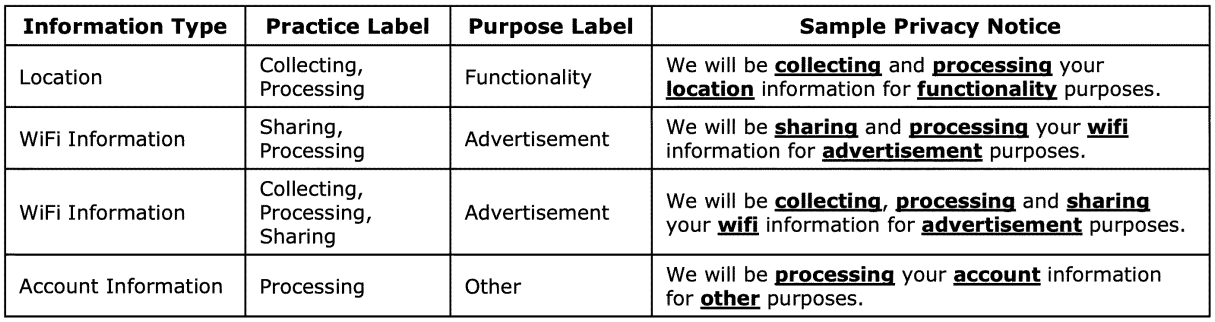
[43] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D. Breaux, and Jianwei Niu. 2018. GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 37–47. https://doi.org/10.1145/3180155.3180196

[44] Shomir Wilson, Florian Schaub, Aswarth Abhilash Dara, Frederick Liu, Sushain Cherivirala, Pedro Giovanni Leon, Mads Schaarup Andersen, Sebastian Zimmeck,

Kanthashree Mysore Sathyendra, N Cameron Russell, et al. 2016. The creation and analysis of a website privacy policy corpus. In *Proc. of the 54th Annual Meeting of the ACL*.
[45] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. 2016. Toward automatically generating privacy policy for android apps. *IEEE Transactions on Information Forensics and Security* 12, 4 (2016), 865–880.
[46] Mu Zhang and Heng Yin. 2016. Automatic Generation of Security-Centric Descriptions for Android Apps. In *Android Application Security*. Springer, 77–98.
[47] Sebastian Zimmeck, Rafael Goldstein, and David Baraka. 2021. PrivacyFlash Pro: Automating Privacy Policy Generation for Mobile Apps. (2021).
[48] Sebastian Zimmeck, Rafael Goldstein, and David Baraka. 2021. PrivacyFlash Pro: Automating Privacy Policy Generation for Mobile Apps. In *28th Network & Distributed System Security Symposium (NDSS 2021) (NDSS 2021)*. Internet Society, Online.
[49] Sebastian Zimmeck, Peter Story, Daniel Smullen, Abhilasha Ravichander, Ziqi Wang, Joel Reidenberg, N Cameron Russell, and Norman Sadeh. 2019. MAPS: Scaling privacy compliance analysis to a million apps. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 66–86.

A APPENDIX

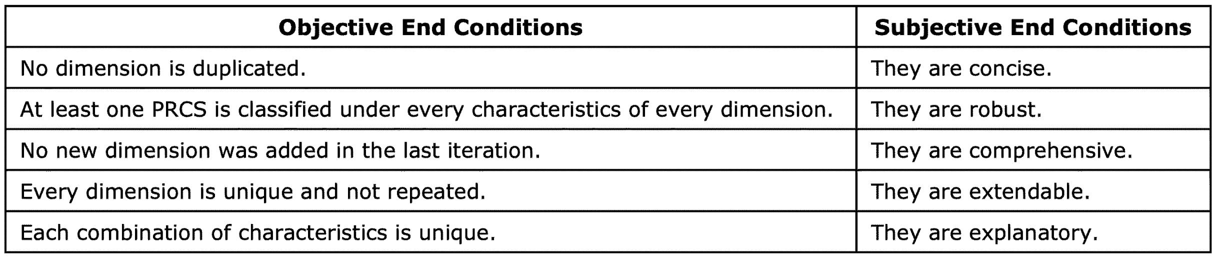
A.1 Examples of Privacy Notices



Information Type	Practice Label	Purpose Label	Sample Privacy Notice
Location	Collecting, Processing	Functionality	We will be **collecting** and **processing** your **location** information for **functionality** purposes.
WiFi Information	Sharing, Processing	Advertisement	We will be **sharing** and **processing** your **wifi** information for **advertisement** purposes.
WiFi Information	Collecting, Processing, Sharing	Advertisement	We will be **collecting**, **processing** and **sharing** your **wifi** information for **advertisement** purposes.
Account Information	Processing	Other	We will be **processing** your **account** information for **other** purposes.

Figure A.1: Example of Sample Privacy Notices that Can be Generated with the Help of Labels Identified by PAcT .

A.2 PAcT Process and its Labels



Objective End Conditions	Subjective End Conditions
No dimension is duplicated.	They are concise.
At least one PRCS is classified under every characteristics of every dimension.	They are robust.
No new dimension was added in the last iteration.	They are comprehensive.
Every dimension is unique and not repeated.	They are extendable.
Each combination of characteristics is unique.	They are explanatory.

Figure A.2: Objective and Subjective End Conditions of Our Taxonomy Development Process [30]

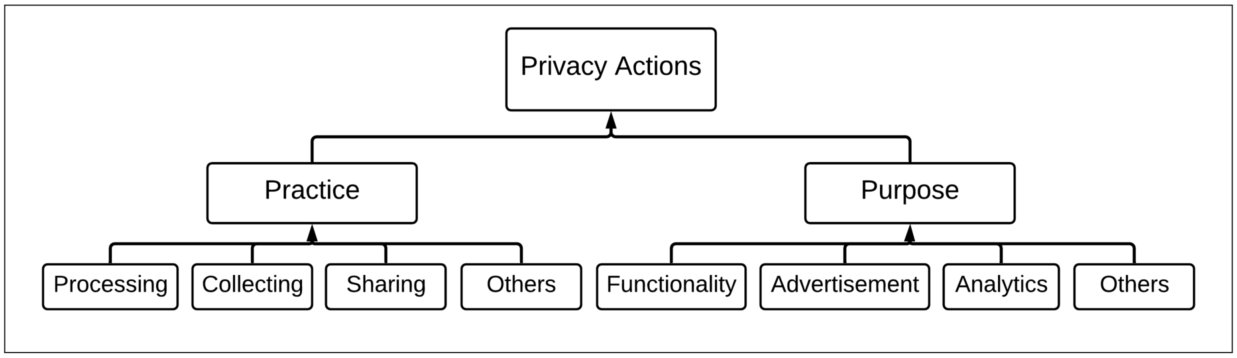
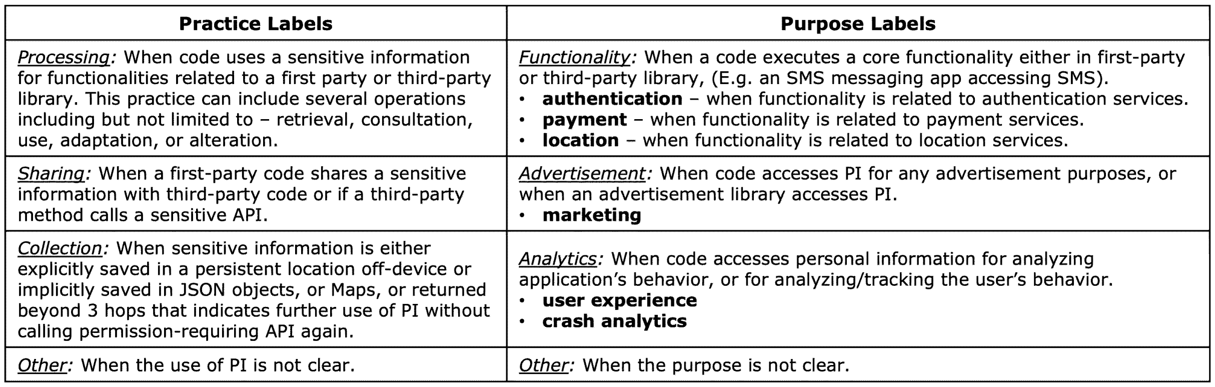


Figure A.3: Hierarchical Breakdown of PAcT.



Practice Labels	Purpose Labels
Processing: When code uses a sensitive information for functionalities related to a first party or third-party library. This practice can include several operations including but not limited to – retrieval, consultation, use, adaptation, or alteration.	*Functionality:* When a code executes a core functionality either in first-party or third-party library, (E.g. an SMS messaging app accessing SMS). • **authentication** – when functionality is related to authentication services. • **payment** – when functionality is related to payment services. • **location** – when functionality is related to location services.
Sharing: When a first-party code shares a sensitive information with third-party code or if a third-party method calls a sensitive API.	*Advertisement:* When code accesses PI for any advertisement purposes, or when an advertisement library accesses PI. • **marketing**
Collection: When sensitive information is either explicitly saved in a persistent location off-device or implicitly saved in JSON objects, or Maps, or returned beyond 3 hops that indicates further use of PI without calling permission-requiring API again.	*Analytics:* When code accesses personal information for analyzing application's behavior, or for analyzing/tracking the user's behavior. • **user experience** • **crash analytics**
Other: When the use of PI is not clear.	*Other:* When the purpose is not clear.

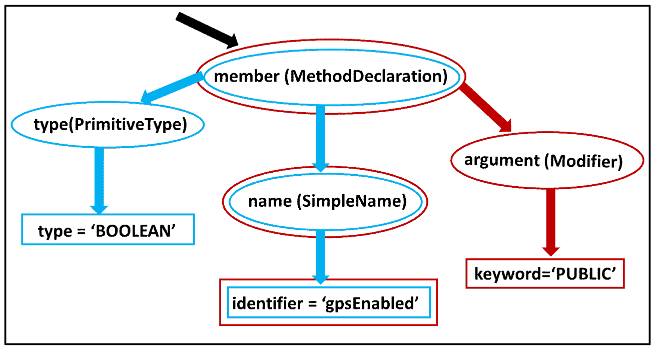
Figure A.4: Definitions of *Practice* and *Purpose* labels, and sub-labels.

A.3 AST Paths

Figure A.5 which shows a PRCS in (a), part of its AST in (b), and selective extracted AST paths in (c). In the tree, (Figure A.5(b)), the terminal nodes are shown in rectangular boxes whereas non-terminal nodes are shown in oval. We also highlight two AST paths in the tree, that are also shown in Figure A.5(c), with blue and red nodes. Each AST path consists of nodes that occur when traversing from one terminal node to another. For example, the first AST path (highlighted in blue in the tree), starts from the terminal node `boolean`, followed by `PrimitiveType`, `MethodDeclaration`, and `SimpleName` non-terminal nodes, and ends at `gpsEnabled` terminal node. In Figure A.5(c), each path is separated by a comma and consists of 3 tokens separated by a space, where each token represents one of the terminal nodes or a concatenation of the non-terminal nodes in the path.

```
public static boolean gpsEnabled(){
    java.util.List v0 = com.foocam.common.android.location
                        .GeoLocationManager.locationManager
                        .getProviders(1);
    boolean v2_3 = v0.iterator();
    while (v2_3.hasNext()) {
        com.foocam.common.android.util.DLog.v(
            new StringBuilder("GPS Loction Provider list contains: ")
            .append(((String) v2_3.next())).toString());
    }
    return v0.contains(gps);
}
```

(a) code sample



(b) Abstract Syntax Tree (AST)

```
boolean PrimitiveType|MethodDeclaration|SimpleName gps|enabled,
public Modifier|MethodDeclaration|SimpleName gps|enabled,
a SimpleName|TypeDeclaration|MethodDeclaration|Modifier public,
public Modifier|MethodDeclaration|PrimitiveType boolean,
a SimpleName|TypeDeclaration|MethodDeclaration|Modifier static,
```

(c) AST Paths

Figure A.5: Code Segment, Abstract Syntax Tree, and AST Paths

A.4 Overview of Privacy Behavior Detection Approach

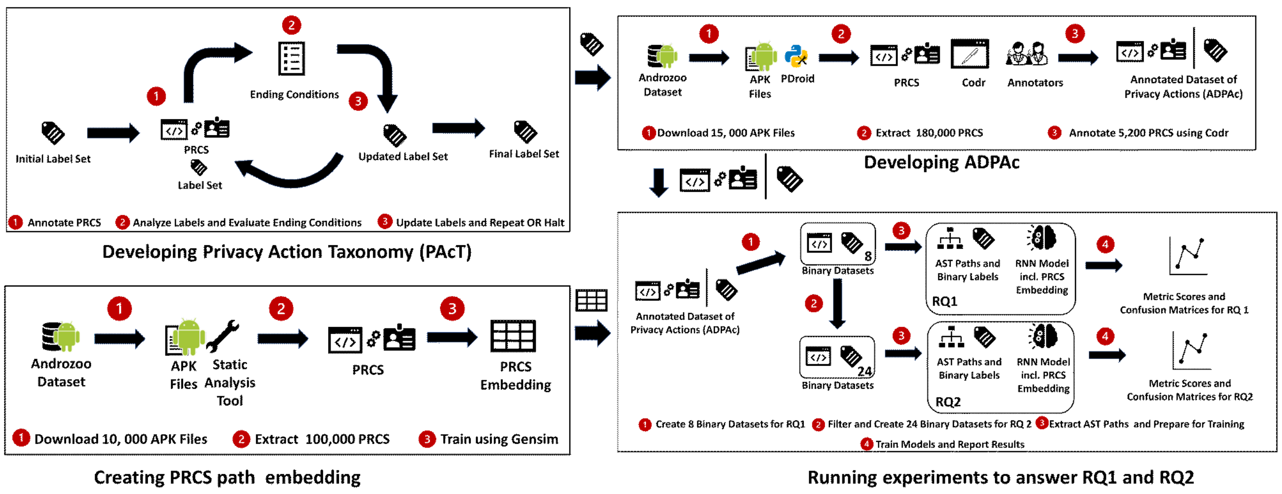

Initial Label Set
PRCS
Label Set
Ending Conditions
Updated Label Set
Final Label Set
1 Annotate PRCS
2 Analyze Labels and Evaluate Ending Conditions
3 Update Labels and Repeat OR Halt
Developing Privacy Action Taxonomy (PAcT)
Androzoo Dataset
APK Files
PDroid
PRCS
Codr
Annotators
Annotated Dataset of Privacy Actions (ADPAc)
1 Download 15, 000 APK Files
2 Extract 180,000 PRCS
3 Annotate 5,200 PRCS using Codr
Developing ADPAc
Androzoo Dataset
APK Files
Static Analysis Tool
PRCS
PRCS Embedding
1 Download 10, 000 APK Files
2 Extract 100,000 PRCS
3 Train using Gensim
Creating PRCS path embedding
Annotated Dataset of Privacy Actions (ADPAc)
8
Binary Datasets
24
Binary Datasets
AST Paths and Binary Labels
RNN Model incl. PRCS Embedding
RQ1
RQ2
Metric Scores and Confusion Matrices for RQ 1
Metric Scores and Confusion Matrices for RQ2
1 Create 8 Binary Datasets for RQ1
2 Filter and Create 24 Binary Datasets for RQ 2
3 Extract AST Paths and Prepare for Training
4 Train Models and Report Results
Running experiments to answer RQ1 and RQ2

Figure A.6: An overview of steps for detecting privacy behavior in code segments: (i) develop PAcT, (ii) develop ADPAc, (iii) create PRCS path embedding, and (iv) conduct experiments.