Towards Fine-Grained Localization of Privacy Behaviors

Vijayanta Jain
University of Maine
vijayanta.jain@maine.edu

Sepideh Ghanavati
University of Maine
sepideh.ghanavati@maine.edu

Sai Teja Peddinti
Google Inc.
psaiteja@google.com

Collin McMillan
University of Notre Dame
cmc@nd.edu

***Abstract*—Privacy labels help developers communicate their application's privacy behaviors (i.e., *how* and *why* an application uses personal information) to users. But, studies show that developers face several challenges in creating them and the resultant labels are often inconsistent with their application's privacy behaviors. In this paper, we create a novel methodology called *fine-grained localization* of privacy behaviors to locate individual statements in source code which encode privacy behaviors and predict their privacy labels. We design and develop an attention-based multi-head encoder model which creates individual representations of multiple methods and uses attention to identify relevant statements that implement privacy behaviors. These statements are then used to predict privacy labels for the application's source code and can help developers write privacy statements that can be used as notices. Our quantitative analysis shows that our approach can achieve high accuracy in identifying privacy labels, with the lowest accuracy of 91.41% and the highest of 98.45%. We also evaluate the efficacy of our approach with six software professionals from our university. The results demonstrate that our approach reduces the time and mental effort required by developers to create high-quality privacy statements and can finely localize statements in methods that implement privacy behaviors.**

***Index Terms*—privacy labels, privacy-behavior, Android applications, machine learning**

1. Introduction

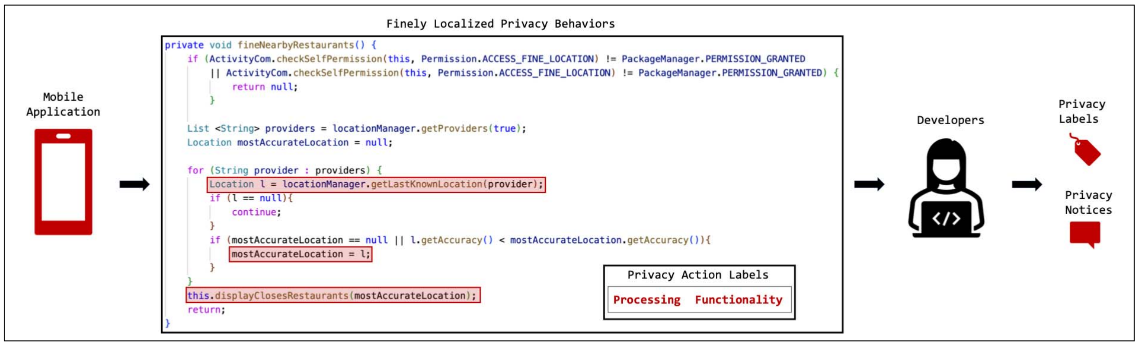
Privacy notices describe *how* an application uses personal information and *why* (i.e., its privacy behaviors), and help users make informed privacy decisions. Application developers are required by privacy regulations [10], [16] and application store policies [43] to provide users with accurate privacy notices. Recently, both application stores (i.e., App Store [1] and Google Play [11]) introduced their versions of privacy "nutrition" labels (or simply privacy labels) [27] to simplify this process. These labels are standardized notice formats that help developers easily describe *how* and *why* their application uses personal information and build trust with the users [31].

Existing challenges in creating privacy *notices* hinder providing accurate *labels*, as well. These challenges range from, difficulty in comprehending privacy behaviors of their applications [31], [42], to gaps in developers' knowledge about privacy concepts [17]. Failure to provide accurate labels violates privacy regulations, which can result in hefty fines for developers [13]. These inaccuracies can also impact users' well-being since it inhibits their ability in making privacy-preserving decisions. Lastly, discrepancies between labels and applications' privacy behaviors also diminish trust between developers and users.

Recent works aim to address the challenges of generating privacy notices, including privacy labels [12], [25], [30], [59], [60]. Some of these efforts leverage static analysis approaches to identify APIs called and use templates [59], questionnaires [60], or developers' annotations [30] to generate notices; while others use machine learning (ML) approaches [24], [25]. For example, Gardner et al. [12] develop *Privacy Label Wiz* which uses static analysis to analyze iOS applications' source code, provides the summary of the results to developers, and prompts them with questions to help create privacy labels for the App Store. Jain et al. [25] create platform-neutral *Privacy Action* labels that describe *how* and *why* an application's code uses personal information. They use a deep learning model to predict these labels from the source code.

While these approaches aid developers in creating privacy labels or notices, they have several drawbacks. For example, Privacy Label Wiz [12] helps understand the privacy behaviors of applications with analysis summary and prompts, but it does not automate the process of creating labels or identifying the purposes. Label creation and purpose identification are still the developers' responsibilities. Moreover, as the application evolves, the tool will rely on developers to keep track of changes in privacy behaviors, including purpose, which may increase the developers' effort. This practice can be especially challenging in settings where developers are part of large teams [31] and the rationale for using personal information is distributed among members. Jain et al. [25] address some of these limitations by automating the process of creating *Privacy Action* labels. However, their approach lacks the necessary source code context which makes it challenging for developers to understand the privacy behaviors of their source code and create privacy notices *solely based on labels*. Consider the code snippet in Figure 1. If developers are only provided with the *Privacy Action* labels `Processing` and `Functionality`, they must read the method to comprehend its behavior and understand how the personal information is `Processed` and for which `Functionality` it is used. The complexity of this task increases when the length of code snippets increases and it spans to include multiple methods/classes.

In this paper, we propose *fine-grained localization* of privacy behaviors. In fine-grained localization, we locate individual statements in source code that encode privacy behaviors and use them to predict their privacy labels.



```
private void fineNearbyRestaurants() {
    if (ActivityCom.checkSelfPermission(this, Permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED
        || ActivityCom.checkSelfPermission(this, Permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
            return null;
        }

    List <String> providers = locationManager.getProviders(true);
    Location mostAccurateLocation = null;

    for (String provider : providers) {
        Location l = locationManager.getLastKnownLocation(provider);
        if (l == null){
            continue;
        }
        if (mostAccurateLocation == null || l.getAccuracy() < mostAccurateLocation.getAccuracy()){
            mostAccurateLocation = l;
        }
    }
    this.displayClosesRestaurants(mostAccurateLocation);
    return;
}
```

Figure 1: Example of how fine-grained localization can help developers create accurate privacy labels and notices.

These localized statements together with predicted privacy labels can then help developers create short sentences to be used in privacy notices. The key novelty of our approach lies in the granularity of locating where privacy behaviors are implemented in source code. While previous localization approaches limit localization to only classes or methods [36], [39], [55] (i.e., coarse-grained localization), our approach moves one step further by also identifying which *statements* within a method implement privacy behaviors (i.e., fine-grained localization). Similarly, static analysis approaches [12], [60] only rely on specific types of statements in a single method, such as API calls, to create privacy notices. Our approach identifies not only API calls but also other statements across multiple methods that use personal information to create privacy labels. Apart from the novelty, our approach provides the benefits of previous efforts such as helping developers understand high-level privacy behaviors (similar to *Privacy Label Wiz* [12]) as well as fine-grained ones and generating privacy labels (similar to Jain et al. [25]) but with higher accuracy (at least ∼11% higher). Our approach is also complementary to existing static and dynamic analysis approaches and provides an *alternative technique* to generate privacy labels. By identifying multiple methods and employing machine learning, our fine-grained localization approach identifies individual statements in those methods providing a much more granular view of privacy behaviors implemented in the application's source code.

We explain how our approach can help create accurate labels and privacy notices with the following example. Consider the same code snippet in Figure 1: If developers are provided with the localized statements (i.e., segments highlighted with red boxes), *along* with the predicted labels of `Processing` and `Functionality`, they can better understand *how location is processed* (i.e., calculating distance) and *for what functionality* (i.e., suggest nearby restaurants) in source code. To provide labels to users, developers can use these predicted labels (i.e., `Processing` and `Functionality`) and map them to their platform-specific labels (App Store or Google Play). They can also use the labels and localized statements to create privacy statements such as "We use your location to calculate the distance and suggest nearby restaurants." and use them in their permission rationales or privacy policies.

To implement our approach, we develop a multi-head encoder model that creates individual representations of multiple methods and uses 'attention' [54] to identify relevant statements in those methods.[1] We demonstrate the efficacy of our approach by training the model on the publicly released ADPAc[2] dataset [25], which contains source code samples and their *Privacy Action* labels. We choose the components of our model by conducting six sets of experiments. In each set, we train a classifier with 24 datasets in the ADPAc, evaluate the optimal model and dataset configurations, and use them for our model. We also analyze the results and provide key insights into how to choose the best combination of model and dataset configurations for identifying privacy behaviors in code.

We evaluate our work both qualitatively and quantitatively. For qualitative evaluation, we recruit six software professionals, with experience in software development and privacy-related research. We ask them to write simple sentences that describe privacy behaviors for code samples with and without fine-grained localization. Our evaluation shows that while there are negligible differences in the statements that the professionals write with/out localization (since they all have privacy expertise), the time and mental effort required are significantly less when the code samples are finely localized. Furthermore, professionals with less experience benefit the most from localization, since they save up to ∼74% of time to write statements of comparable quality and details to those written by the most experienced ones. Quantitatively, we use accuracy and F-1 scores to gauge the model's performance in predicting *Privacy Action* labels. Our evaluation shows our model increases the baseline accuracy [25] by at least 11% with up to 30% for some labels. Our lowest accuracy is 91.41% and the highest is 98.45% across labels. We also measure the accuracy of fine-grained localization by asking three of the six software professionals who are more experienced in software development and privacy to manually inspect the statements in code samples that are highlighted by the attention module. The results show that at least one annotator agrees that 85% of statements implement privacy behaviors. Our analysis strongly demonstrates that our approach can identify privacy labels and help in writing high-quality privacy statements.

1. A recent NLP work has raised questions about an attention module's capability to identify relevant parts of input sequences [23]. However, several studies indicate that attention is important [49], [51] and can be used for interpreting the results of a classification task. Hence, we use attention to select relevant statements and localize privacy behaviors.
2. https://github.com/PERC-Lab/PAcT

In summary, our main contributions are as follows:

1) We address the issue of creating accurate privacy labels by developing automated fine-grained localization to identify statements in methods that implement privacy behaviors and that explain *how* and *why* personal information is used. This approach extends the granularity of localization in previous works to individual statements.
2) We implement our approach by developing a novel attention-based deep learning model, for which we conducted six sets of experiments with 24 datasets to meticulously choose the best model and dataset configurations. We share insights from these experiments to help researchers use them as a blueprint for classifying privacy behaviors in source code.
3) Our evaluation demonstrates that our approach establishes a new state-of-the-art for predicting *Privacy Action* labels, its efficacy in finely localizing privacy behaviors, and helping create privacy statements by reducing time and effort.

2. Related Work

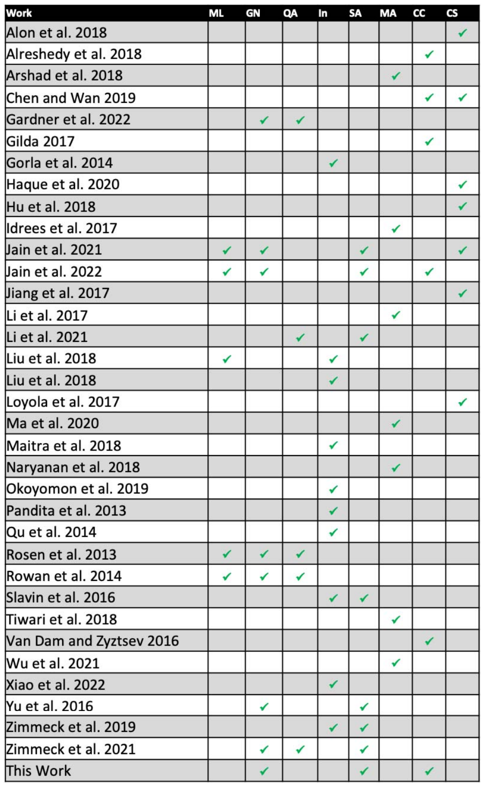
Figure 2 summarizes the prior work by showcasing trends in finding discrepancies between privacy notices and application source code, creating privacy notices, identifying malicious applications, and classifying and summarizing code segments.

2.1. Finding Discrepancies

A significant effort has been made to identify discrepancies between an application's privacy behaviors and its privacy notices [15], [32], [37], [40], [50], [56], [61]. Most works focus on matching the application's privacy behaviors with its privacy policy [37], [61], application description [15], [41], [44], permission rationales [32], or privacy labels [56], using static analysis, and counting the instances of discrepancies. These works have identified significant inconsistencies between applications' privacy behaviors and their notices and *highlighted* the issue. Our work aims at *resolving* these discrepancies by generating accurate privacy labels and localizing privacy behaviors in source code to help create consistent and detailed notices.

2.2. Creating Privacy Notices

A number of studies focus on creating privacy notices [12], [24], [25], [30], [46], [47], [59], [60]. These approaches either create privacy policies using questionnaires [46], [47], [60] (similar to privacy policy generators), or provide notices in different formats, such as permission rationales [33], privacy describing statements [59], or in-application notices [30]. AutoPPG [59], PrivacyFlash Pro [60], Privacy Label Wiz [12], Honeysuckle [30], and PAcT [25] are closely related to our work since they also aim to aid developers to create privacy notices using source code. AutoPPG [59] analyzes the APIs called to identify the personal information used and then uses a static `subject form object [condition]` template to create privacy statements, but it lacks the rationale



Work	ML	GN	QA	In	SA	MA	CC	CS
Alon et al. 2018								✓
Alreshedy et al. 2018							✓	
Arshad et al. 2018						✓		
Chen and Wan 2019							✓	✓
Gardner et al. 2022		✓	✓					
Gilda 2017							✓	
Gorla et al. 2014				✓				
Haque et al. 2020								✓
Hu et al. 2018								✓
Idrees et al. 2017						✓		
Jain et al. 2021	✓	✓			✓			✓
Jain et al. 2022	✓	✓			✓		✓	
Jiang et al. 2017								✓
Li et al. 2017						✓		
Li et al. 2021			✓		✓			
Liu et al. 2018	✓			✓				
Liu et al. 2018				✓				
Loyola et al. 2017								✓
Ma et al. 2020						✓		
Maitra et al. 2018				✓				
Naryanan et al. 2018						✓		
Okoyomon et al. 2019				✓				
Pandita et al. 2013				✓				
Qu et al. 2014				✓				
Rosen et al. 2013	✓	✓	✓					
Rowan et al. 2014	✓	✓	✓					
Slavin et al. 2016				✓	✓			
Tiwari et al. 2018						✓		
Van Dam and Zyztsev 2016							✓	
Wu et al. 2021						✓		
Xiao et al. 2022				✓				
Yu et al. 2016		✓			✓			
Zimmeck et al. 2019				✓	✓			
Zimmeck et al. 2021		✓	✓		✓			
This Work		✓			✓		✓	

Figure 2: Selection of closely-related, peer-reviewed publications. Column *'ML'* = ML based; *'GN'* = Generate Notices; *'Q/A'* Question-Answering based; *'In'* = Inconsistency Analysis; *'SA'* = Static/Dynamic Analysis; *'MA'* = Malicious Applications; *'CC'* = Code Classification and *'CS'* = Code Summarization.

for using personal information. PrivacyFlash Pro [60], Honeysuckle [30], and Privacy Label Wiz [12] provide rationales in their notices, however, they rely on developers' efforts to do so. Jain et al. [25] automatically provide *Privacy Action* labels that describe *how* and *why* personal information is used. However, as discussed in Section 1, these labels lack source code context; adding more work for developers to understand their privacy behaviors. In this paper, we extend these efforts, by automating the process of creating accurate privacy labels and providing developers with source code context to help them understand the privacy behaviors of their applications.

2.3. Identifying Malicious Applications

Studies that identify malicious mobile applications are tangential to our work; however, there are some similarities in our approaches that are worth mentioning. Several works have provided approaches to identify malicious applications [5], [22], [52] which pose security risks. Recently, some studies extended these efforts to localize malicious code in applications [29], [36], [39], [55]. These approaches use program graphs, such as call or dependency graphs, to represent an application, extract features, and classify them using an attention-based deep learning model [36], [39], [55]. Using attention weights, these approaches identify relevant features used for classification, which they use to localize malicious source code. Our approach to localizing privacy behaviors in source code follows a similar logic. However, a significant difference between our approaches is the granularity of localization. These approaches limit localization to malicious packages [29] or methods [36], [39], [55] (i.e., coarse-grained localization), whereas, in our approach, we localize statements within a method which provides a more fine-grained localization of privacy behaviors.

2.4. Classifying and Summarizing Code

Our work also draws inspiration from efforts in software engineering to classify and summarize code. Classification tasks in this field primarily focus on detecting the programming language of code snippets [3], [14], [53], whereas summarization tasks focus on transforming code snippets into natural language text for various tasks. The format of the generated text differs based on the purpose, such as transforming code differences into commit messages for version control [26], [35], or into comments for documentation [2], [9], [19], [48]. Our work extends these efforts to comprehend code and modifies it to finely localize privacy behaviors and predict their privacy labels.

3. Background

3.1. Permissions and Static Analysis

Mobile operating systems, such as Android, employ a permissions system to allow developers to access users' personal information while giving control to users to protect their personal information. In this system, if developers want to access personal information, they call a system API [3] and declare necessary permissions, such as in `AndroidManifest.xml` file for Android applications. When the application requires access to sensitive information, users are asked if they would permit the application to access this information, thereby giving them control over their personal information.

Previous work uses static analysis to identify system API calls and extract methods that call them [25]. They refer to these extracted code snippets as *Permission-Requiring* Code Segments (PRCS) since they call APIs that require permission (i.e., *permission-requiring* APIs). Since a method calling permission-requiring APIs can share the accessed information with other methods for further use, a PRCS includes multiple methods linked via a call graph. Each PRCS segment contains at most three methods because they found that in ∼80% of cases, personal information is used within these three methods [24], [25]. Since personal information can "hop" between methods in a PRCS, each method is referred to as a "hop". The first hop in a PRCS is the method that calls the permission-requiring API and each PRCS contains at least this first hop. The subsequent methods are called the second and third hops, respectively (we interchangeably use "hop" and "method"). In our approach, we predict labels and localize privacy behaviors for PRCS that consist of up to three hops. Applications can also access and use personal information via user interfaces [4], [21], [38]. However, in this work, we limit the scope of code segments to the ones that call system APIs, since our goal is to demonstrate the feasibility of fine-grained localization and not to show the coverage of extracting all source code.

3. https://developer.android.com/reference/

3.2. Privacy Action Taxonomy and Dataset

Privacy Action Taxonomy (PAcT) is a taxonomy that defines privacy behaviors implemented in source code [25]. The goal of this taxonomy is to help consistently detect privacy behaviors in the application's source code and create privacy labels for them. In this taxonomy, there are two categories of labels: *Practice* and *Purpose*. The labels in the *Practice* category describe *how* a code segment uses personal information whereas the labels in the *Purpose* category answer *why*. Both categories contain 4 labels each. The *Practice* category contains `Processing`, `Collecting`, `Sharing`, and `Other` labels. Whereas the *Purpose* category contains `Functionality`, `Advertisement`, `Analytics`, and `Other` labels. The definition of each label is described in Jain et al. [25].

Jain et al. [25] used PAcT to create an annotated dataset (called ADPAc) of code segments and their *Privacy Action* labels. This dataset contains ∼5,200 PRCS and ∼14,000 labels, which is publicly available (see Section 1). Since each code segment can implement multiple behaviors, some samples are annotated with multiple labels; hence, ADPAc is a multi-class multi-label dataset. ADPAc also provides binary datasets for these labels. These binary datasets include both positive and negative samples; each positive sample corresponds to the presence of a specific privacy behavior whereas a negative sample corresponds to its absence. For example, in the binary dataset of `Collecting` label, a positive code sample implements collecting behavior whereas a negative code sample does not. Each code sample is represented as an Abstract Syntax Tree (AST) containing paths within, where a path is a traversal of nodes in an AST. AST and its paths are explained in more detail in Appendix A.

Additionally, as mentioned earlier, each code sample can include up to three hops. Therefore, for each label, there are three versions of the binary dataset, which include the same samples containing different numbers of hops. For example, `Collecting_1_Hop`, `Collecting_2_Hop`, and `Collecting_3_Hop` have the same samples but contain one (the first hop), two (the first two hops), and three (all the three) hops, respectively. Since there are eight *Privacy Action* labels in PAcT and each has three versions, there are 24 datasets in total. In this work, we use these 24 binary datasets for our experiments and for training our multi-head encoder model.

3.3. Attention

The concept of attention was introduced by Bahdanau et al. [6] as a method to jointly align and translate sequences which significantly improves the translation tasks. Subsequently, attention-based networks were used in several other NLP tasks and achieved state-of-the-art performance [34], [57], [58]. The idea behind attention is to quantitatively identify tokens in the input sequence that are more relevant to the task than other tokens. There are different variations of attention [54]. In this work, we use self-attention (or intra-attention); a mechanism in which the weights of the input tokens are determined based on the importance of other tokens in the same sequence.

4. Approach

We now explain our approach to localizing privacy behaviors in application source code and describe the implementation details of our model. In summary, our approach works as follows: (i) extract AST paths for each hop in a code sample, (ii) embed the AST paths, (iii) encode embedded AST paths, (iv) use attention to identify relevant paths, (v) use fully connected layers to predict *Privacy Action* label, (vi) extract relevant paths identified by the attention module, and (vii) map these paths to source code to finely localize privacy behaviors. Figure 3 (a) shows a detailed overview of our approach.

4.1. Towards Fine-Grained Localization

We provide fine-grained localization of privacy behaviors by using attention weights to quantify relevant paths in each hop and then mapping these paths to source code. Towards this approach, the first step is to extract AST paths from each hop of a code sample. As described in Section 3.2, each code segment (PRCS) extracted from an application consists of three methods linked via a call graph. In line with other research studies that classify/summarize source code [2], [9], we represent each code sample using AST paths. We extract these paths for each hop using a tool called "`astminer`" [28].

The second and third steps are to embed and encode the extracted AST paths. Recall that each AST path consists of terminal and non-terminal nodes (as described in Appendix A). To embed each path, we first tokenize the paths into individual (terminal and non-terminal) nodes. We choose to tokenize non-terminal nodes based on our experimental results, as explained in Section 6.1.1. Next, we embed each node using a pre-trained embedding model that we created using Gensim [45]. The combined embedding of each node in an AST path is then passed to recurrent layers for encoding. To encode each hop, we use separate encoder heads to ensure that we preserve the semantic differences between the three methods. Since there are three methods in each sample, there are three heads in the multi-head encoder model. These multiple encoders are an upgrade from the baseline approach that used a single encoder head to encode all three hops.

In the fourth step, we need to identify relevant paths that implement privacy behaviors. This is necessary to predict *Privacy Action* labels and finely localize privacy behaviors in the subsequent steps. To identify these relevant paths, we use attention. In each encoder head, an attention module uses a weighting mechanism that provides attention weights for each path in each hop to quantify the relevance of each encoded path. This relevance indicates if a path implements a privacy behavior or not, i.e., paths with higher attention weights most likely use personal information which may help predict a label.

In the fifth step, we predict *Privacy Action* labels. We combine the attention weights with their respective encoded paths and then pass them to the fully-connected layers. This step creates a weighted representation of each *hop* and provides the fully-connected layers to "comprehend" the privacy behaviors implemented across three hops. Using non-linearity, the output of the layers is converted into a prediction for a label. In the second last step towards fine-grained localization, we extract the output of the attention module (i.e., the attention weights) from each head and match them with their corresponding paths in each hop. This gives us the quantified relevance of each *AST path* in each hop. We then sort these paths based on their weights and select 20 paths with the highest weights from each hop. We select this number based on experimental results that we explain in Section 6.3.

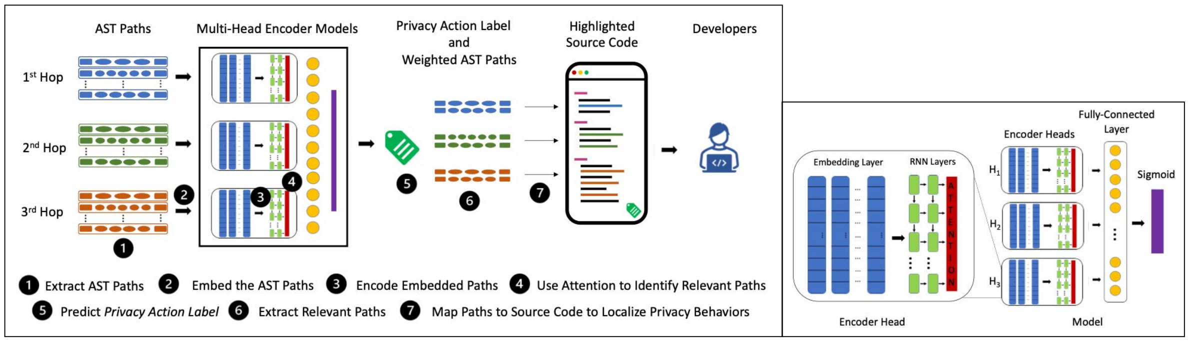
Lastly, we map these 20 paths from each hop to the source code using an automated script as follows: the script inspects the terminal nodes of each path, which contain the name of the identifier, and maps it to the line in the source code that contains the identifier. For example, in Appendix A - Figure 8 (c), the first AST path has a terminal node "`getLastKnownLocation`". Since the corresponding source code (Figure 8 (a)) has the method name `getLastKnownLocation`, the first AST path is mapped to the highlighted statement in Figure 8 (a).

These mapped statements implement privacy behaviors and are highlighted in the code, thereby finely localizing privacy behaviors. These seven steps together provide an *automated* fine-grained localization mechanism.

We design our localization approach to identify privacy-relevant snippets at statement-level granularity since it provides maximum precision. A larger granularity, say a block of code comprised of several lines, may work in some cases where privacy-relevant statements are written together. In cases where privacy-relevant statements are spread out, this block-level granularity will highlight several non-relevant statements and result in a less precise localization. A statement-level granularity, thus, is a better approach since it identifies privacy-relevant code that are either written together or they are spread out.

4.2. Implementation Details

We now explain the implementation details of our steps: After we extract the AST paths for a code sample C, we randomly select num_paths paths. We decide how many paths to select from each hop based on our experiments (see Section 6.1.3). Next, we tokenize the nodes in AST paths. Each AST path p_i can be represented as $p_i = [t_s; t_N; t_e]$, where t_s and t_e are the start and terminal nodes, and $t_N = [t_1; ...; t_8]$ is a list of non-terminal nodes. We found that in the ADPAc dataset, each sequence of non-terminal nodes can be of max length 8. Therefore, after tokenizing, each path p_i can be represented as a list of 10 terminal and non-terminal nodes, $p_i = [t_s; t_1; ...; t_e]$. In case the length of non-terminal nodes is less than 8, we pad the AST path with zeros. We also pad each path with a zero to denote the end of the path. Hence, each tokenized path p_i is 11 tokens long (the last token is padding). Since, each hop h_i is represented by num_paths AST paths, $h_i = [p_1, p_2, ...p_{num\_paths}]$, where $i \epsilon [1, 2, 3]$. Mathematically, each hop h_i is a $num\_paths \times 11$ matrix. Each



(a) An overview of our approach to classify and localize privacy behaviors. (b) Detailed architecture.

Figure 3: Overview of our approach and detailed architecture of multi-head encoder model

code segment C contains three hops, i.e., $C = [h_1; h_2; h_3]$, which makes it a $3 \times num\_paths \times 11$ tensor.

As shown in Figure 3 (b), each encoder head (H_1, H_2, and H_3) contains an embedding layer E, two recurrent layers, and an attention module. After tokenizing each path, we encode them: each individual hop h_i in C is passed to an encoder head H_i. Inside each head, each hop is embedded as $E_{h_i} = [E_{p_1}; E_{p_2} ... E_{p_{num\_paths}}]$, where E_{h_i} is a $num\_paths \times 11 \times embed\_size$ tensor containing embedded AST paths. Each embedded path E_{p_i} is represented as $E_{p_i} = [E_{t_s}; E_{t_1}; ...; E_{t_e}]$, where E_{p_i} is a $11 \times embed\_size$ matrix and E_{t_x} is the embedding of an individual node in an AST path which is an $embed\_size$ dimensional vector.

After embedding, we pass E_{h_i} through the recurrent layers for encoding and use the output L_i as the context and $hidden_i$ as the hidden state i.e., $L_i, hidden_i = RNN(E_{h_i})$. Here L_i is $num\_paths \times embed\_size$ matrix and the hidden state is a num_paths size vector. We pass the hidden states to the attention module which returns attention weights of size num_paths. We stack the attention weights from the three heads which give us a $3 \times num\_paths$ matrix. We pass this matrix through fully-connected layers and apply sigmoid non-linearity to get the classification probability. Lastly, we match the attention weights from each hop to the AST paths and use a script to map them to their source code.

5. Experiments

In this section, we describe the rationale for our experiments, their setup, and evaluation techniques.

5.1. Research Questions

Our research objective is threefold: first, to find the optimal model configurations for classifying privacy behaviors in code snippets. Second, evaluate our model's performance in classifying privacy behaviors in comparison with other models. Third, evaluate the feasibility of our approach to finely localize privacy behaviors and its efficacy in helping to write privacy statements. For these objectives, we ask the following three research questions:

RQ 1: Which configurations provide the optimal performance for classification of *Privacy Action* labels?

Based on the specific configuration, we ask the following sub-research questions:

RQ 1.1: How does the tokenization of non-terminal nodes affect the classification performance of the model?

RQ 1.2: Which type of recurrent layers perform better? LSTM or Bi-LSTM?

RQ 1.3: What is the optimal number of AST paths to represent each code sample?

RQ 2: Does the multi-head encoder model provide any quantitative increase in the classification performance as compared to other models?

RQ 3: Is our fine-grained localization efficacious?

We evaluate the efficacy of our approach to localize privacy behaviors and help write privacy statements qualitatively and quantitatively based on the following sub-research questions:

RQ 3.1 How does fine-grained localization help developers write privacy statements?

RQ 3.2 What is the accuracy of fine-grained localization in identifying privacy behaviors?

The rationale for RQ 1 is based on the following two reasons: first, the baseline model [25] did not include many common model configurations that could improve the model's classification performance. Second, there is no systematic study that compares how the *combination* of each of these components affects the model's performance for code classification. Therefore, we experiment with tokenization, Bi-LSTM layers, and the number of AST paths. We choose these configuration options for the following reasons. *Tokenization (RQ 1.1):* In the baseline approach, AST paths are tokenized into three tokens where the terminal nodes are considered as two tokens and the non-terminal nodes are considered as a single token (see Appendix A for reference). By not tokenizing non-terminal nodes, the syntactic structural details of each code sample are diminished which could potentially hinder the classification accuracy. *Bi-LSTM Layers (RQ 1.2):* The primary difference between LSTM and Bi-LSTM layers is the additional backward direction encoding of input sequences in Bi-LSTM layers which provides context surrounding each token in the sequence. Since an AST path can be traversed in either direction, including a reversed encoding may improve the classification accuracy. *AST Paths (RQ 1.3)*: The number of paths used to represent a code sample is varied from 100 - 300 in code summarization studies [20], [48]. Therefore, to evaluate the optimal number of

AST paths, especially for classification, we compare the performance of 100, 200, and 300 AST paths. It is possible to experiment with several other configuration choices, such as source code tokens versus AST paths, to learn their effects on a classification task; however, these are tangential to the goal of this paper since we focus on the feasibility of fine-grained localization.

In RQ 2, we aim to quantitatively evaluate whether our novel multi-head encoder model improves the performance of classifying *Privacy Action* labels in comparison to the baseline model and its derivatives with modified configurations (i.e., models from RQ 1).

In RQ 3, we evaluate fine-grained localization using qualitative (RQ 3.1) and quantitative (RQ 3.2) approaches. Qualitatively, we first analyze automated localization and then evaluate its dis/advantages in helping software professionals write privacy statements. Quantitatively, we evaluate the accuracy of our approach in identifying privacy behaviors in source code.

5.2. Experimental Setup

We use the 24 binary datasets from ADPAc [25] to answer RQ 1 (*Exp 1-6*). Recall that each of the 8 *Privacy Action* labels contains 3 versions of the dataset (for example, `Collecting_1_Hop`, `Collecting_2_Hop`, and `Collecting_3_Hop`), with each version containing paths from one more hop than the previous version (see Section 3.2). To answer RQs 1.1-1.3, we begin with the baseline model [25] and add attention (i.e., *Exp 1: L_100*). Based on the experiments, we make further configuration changes to this attention-based model. We intentionally add attention to the baseline model, since it is key to our approach and several studies demonstrate the efficacy of attention in improving the models' performance across different NLP tasks [6], [54].

To limit the number of explored configurations that help us answer our RQs, we make incremental changes to determine the impact of each configuration and identify optimal selection rather than trying out all possible configuration combinations. We modify one configuration in each experiment, and based on the results, we keep this configuration fixed for subsequent experiments. For example, based on the results for RQ 1.1 (i.e., *Exp 1: L_100*), we make a decision to include/not-include the tokenization of non-terminal nodes. Then, when we evaluate RQ 1.2 and RQ 1.3, we use that configuration and only vary the choice of recurrent layer and the number of AST paths.

To set up the experiments for RQ 1.1, (*Exp 1: L_100*), we tokenize each node in an AST path, including non-terminal nodes. For RQ 1.2, (*Exp 4: Bi_100*), we replace LSTM layers with Bi-LSTM in the model. For RQ 1.3 (*Exp 1-6*), we randomly pool N AST paths from each code sample for each version of the dataset. For example, if $N = 100$, for `Collecting_1_Hop` dataset, which contains AST paths from only the first hop, we randomly select 100 AST paths from each sample. For `Collecting_2_Hop` dataset, we pool 100 AST paths from the combined paths of the first two hops, and similarly, we pool 100 paths from the combined paths of all three hops in `Collecting_3_Hop` case. In these experiments, N varies between 100 - 300 at 100-step increments. In case a sample contains less than N paths,

TABLE 1: RQ 1 and RQ 2 experiment configurations. "Tok": tokenizing non-terminal nodes. "Attn": using attention.

	Tok	**Attn**	**RNN Type**	**Paths**
Baseline	False	False	LSTM	100
Exp 1: L_100	True	True	LSTM	100
Exp 2: L_200	True	True	LSTM	200
Exp 3: L_300	True	True	LSTM	300
Exp 4: Bi_100	True	True	Bi-LSTM	100
Exp 5: Bi_200	True	True	Bi-LSTM	200
Exp 6: Bi_300	True	True	Bi-LSTM	300
Multi-Head Encoder	True	True	LSTM	100/hop

we pad the sample with zeros, which we call null paths. We summarize these various experiment configurations explored in RQ 1 (and its sub-questions) in Table 1.

To answer RQ 2 (*Multi-Head Encoder*), we train our multi-head encoder model and compare its performance with the baseline model (i.e., Jain et al. [25]) as well as the best configurations derived from RQ 1 (Table 1). Note that the RNN type for the multi-head encoder model in Table 1 applies to each encoder head. To prepare the dataset, we extract the AST paths from 1_Hop, 2_Hop, and 3_Hop datasets for each *Privacy Action* label and separate the paths for each hop. For example, from `Collecting_1_Hop`, `Collecting_2_Hop`, and `Collecting_3_Hop` datasets, we separate the paths belonging only to the first hop, second hop, and third hop, respectively. Since this model is trained using paths from all three hops, we only compare the results with the 3_Hop versions of the baseline and RQ 1 experiments (i.e., *Exp 3 and Exp 6*).

We trained the models for RQ 1 and 2 using the following hyperparameters: a batch size of 8 and the Adam optimizer to modify the weights. The learning rate was fixed at 1e-5 and we used binary cross entropy to penalize the model. For each dataset, training, validation, and test sets were split in an 80:10:10 ratio. We chose these hyperparameters since these were also used in the baseline model [25]. For RQ 1, each model was trained for 50 epochs since we did not find improvement in the results after 50. For RQ 2, we varied the number of epochs since the model convergence differed for each label. For both RQs, after each epoch, the models were evaluated on the validation set. If they achieved better accuracy than the previous best epoch then the parameters were saved. We also monitored the training and validation accuracy to ensure the models did not overfit. After the training, the parameters from the best epoch were used on the test set for evaluation. We answer RQ 1 and 2 based on quantitative metrics, accuracy and F-1 scores. These are standard metrics used in classification tasks to compare the performance of different ML models. We balance our comparison by discussing overall patterns in classification accuracy for each configuration change while also highlighting interesting changes in the results of individual labels. To develop the model, we use PyTorch 1.8.1 with Python 3.7 and run all experiments on a workstation with a Xeon CPU, 54 GB RAM, and a Tesla T4 GPU.

To answer RQ 3, we first conduct a manual inspection of mappings of some samples to analyze and draw insights into our approach to finely localize privacy behaviors. Next, to answer the two sub-research questions, we select

20 random samples from the test set of all labels, use our script to localize privacy behaviors in them, i.e., map the AST paths with the highest attention weights to the source code, and highlight them.

To qualitatively evaluate fine-grained localization in RQ3.1, we asked six software professionals at our university to write privacy statements for each sample, explaining *how* the personal information is being used and *why*. To investigate any differences in the statements written when the samples are localized versus when they are not, we divided the six annotators (i.e., software professionals referred to as annotators from here onward), into two groups based on their experience. Annotators in Group #1 (i.e., Annotators #1, #2, and #3), have more than four years of software development experience and two years of experience in privacy research. Whereas Group #2 annotators have about two years of software development experience and approximately one year of experience in privacy research. Each group was given 20 samples, of which only 10 were finely localized. Moreover, the samples that were localized for Group #1 were not localized for Group #2, resulting in privacy statements for the same sample that were written with and without localization.

Apart from writing privacy statements, we also asked each annotator if and how the localization helped, and the time it took them to write each statement. We compare privacy statements and the time taken between localized and non-localized samples of similar length for each annotator as well as between the two groups for the same samples. These comparisons help us understand how fine-grained localization affects annotators individually, as well as among different populations (i.e., professionals with little privacy experience vs. those with more experience).

In RQ 3.2, we evaluated the accuracy of fine-grained localization by using the same 20 samples from RQ 3.1, all of which were finely localized. We then asked annotators in Group #1 who are privacy experts to evaluate if the highlighted statements implement privacy behaviors by providing a binary response ('yes'/'no') for each highlighted statement. Since each sample consists of three methods (see Section 3), we evaluated a total of 60 methods where 230 of their statements were highlighted. To measure the inter-rater agreement among the annotators we used Krripendorff's Alpha[4] and Fleiss's Kappa[5].

6. Results

In this section, we report our results and answer our research questions. Table 2 shows the quantitative results for RQ 1 and RQ 2, which includes the baseline results [25] in the first column (*Baseline*). We show the confusion matrices for RQ 2 in Appendix B - Figures 9 and 10. For RQ 3, we first discuss automated fine-grained localization with one representative code sample shown in Figure 4, and its weighted AST paths in Figure 5. We, then, discuss dis/advantages of our approach in helping software professionals write privacy statements with a representative sample in Fig. 6 (additional examples are in Figure 14-15 in Appendix E). Lastly, we report the accuracy of our approach for RQ 3.2 (examples shown in Figures 16 and 17 in Appendix F).

4. https://www.statisticshowto.com/krippendorffs-alpha/
5. https://www.statisticshowto.com/fleiss-kappa/

6.1. RQ 1: Optimal Configurations

As stated in Section 5, to answer sub-research questions, we make incremental changes to the configuration starting with the baseline model. Overall, we find an attention-based model with tokenization of non-terminal nodes helps significantly improve the performance. Between LSTM and Bi-LSTM layers, the difference is insignificant when we use fewer paths (say 100), but as we increase this number (to say, 300), Bi-LSTM provides better results. 300 paths are an optimal choice to represent each code sample, and 100 paths to represent each hop.

6.1.1. RQ 1.1: Tokenization of Non-Terminal Nodes. The results for RQ 1.1 (*Exp 1:L_100* column in Table 2) indicate that by tokenizing non-terminal nodes (and using attention), we noticeably increase the accuracy for most labels. We observe that accuracy increases by $\sim$5% on average for *Practice* and *Purpose* labels with the F-1 score also increasing for most labels. For some labels, the increase in accuracy is more significant; for example with `Processing_1_Hop` and `Advertisement_1_Hop`, the accuracy increases by 12% and 15%, respectively. These improvements in the scores can be attributed to the increase in the syntactic structural information of a code sample, which is what non-terminal nodes represent.

For example, a code sample '`Processing`' personal information, say location, executes several operations ranging from *comparing* accuracy of location providers to *calculating* distance to the user's address. The syntactic structural information of these operations is captured by the tokenized non-terminal nodes, and when provided to the model, it can help predict `Processing` with better accuracy. Additionally, attention helps the model learn which structures contribute to `Processing` and which ones do not. To test the efficacy of attention, we ran an experiment in which we tokenized non-terminal nodes and trained using the baseline model (i.e., without attention) [25]). In this experiment, we noticed an average accuracy of 59.85% and 54.40% for *Practice* and *Purpose* labels, which are 7% and 20% lower than the baseline models (i.e., *Baseline* column in Table 2). This noticeable decrease in the performance indicates the significance of attention, especially, with tokenized non-terminal nodes.

Interestingly, we found a noticeable decrease in F-1 scores for `Sharing` in *Exp 1:L_100*. This decrease is most likely due to an information overload of syntactic structures. Since '`Sharing`' often occurs with calls to third-party libraries or third-party libraries calling permission-requiring APIs [25], such information is embedded in the identifiers of source code, i.e., the terminal nodes of AST paths. By increasing syntactic structural information (via tokenizing non-terminal nodes), the identifier information present in nodes decreases disproportionately, making it difficult for the model to predict this label. This is also evident by the inverse classification performance between `Processing` and `Sharing` labels, i.e., when *Processing* is better predicted (*Exp 1:L_100*) with tokenized non-terminal nodes, *Sharing* is not and vice versa (*Baseline*). As noted in the baseline, *Sharing*

TABLE 2: Accuracy and F-1 Scores for each experiment conducted for RQs 1 and 2. All experiments except the baseline, tokenize non-terminal nodes and use attention. We use "L" to denote use of LSTM layers, "Bi" for Bi-LSTM layers, and "100—200—300" are the number of AST paths used in the experiment. See Table 1 for details. The color of the cell denotes the range of the score. Red: 50-60%, Orange: 60-70%, Lime: 70-80%, Light Green: 80-90%, Green: 90+%. A row at the end of each category shows the average score for all datasets in that category.

Dataset	Baseline		Exp 1: L_100		Exp 2: L_200		Exp 3: L_300		Exp 4: Bi_100		Exp 5: Bi_200		Exp 6: Bi_300		Multi-Head Encoder	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
								Practice								
Collecting																
1 Hop	70.39%	74.58%	75.00%	72.46%	73.03%	70.07%	72.37%	69.12%	73.68%	71.01%	74.34%	71.94%	72.37%	69.12%	-	-
2 Hop	68.42%	73.03%	74.34%	71.53%	75.66%	73.38%	75.66%	74.48%	75.66%	73.38%	76.97%	74.82%	84.87%	91.81%	-	-
3 Hop	71.09%	74.36%	76.32%	73.53%	76.32%	73.53%	75.66%	73.38%	76.32%	73.13%	77.63%	73.85%	84.87%	91.81%	98.03%	98.04%
Sharing																
1 Hop	65.10%	69.82%	66.41%	54.74%	67.97%	53.23%	67.71%	56.64%	66.67%	52.59%	66.67%	55.24%	63.02%	77.10%	-	-
2 Hop	67.10%	73.39%	66.41%	52.40%	69.27%	57.55%	68.75%	56.20%	66.93%	51.71%	67.97%	55.91%	64.58%	75.89%	-	-
3 Hop	71.09%	77.58%	68.23%	55.15%	67.71%	55.71%	68.49%	56.94%	68.49%	55.68%	68.23%	57.64%	64.32%	77.36%	91.41%	90.32%
Processing																
1 Hop	58.75%	51.04%	70.50%	74.89%	69.75%	71.93%	71.75%	73.16%	71.00%	75.93%	71.50%	72.73%	71.50%	79.20%	-	-
2 Hop	64.75%	59.83%	72.50%	76.89%	71.00%	77.17%	72.25%	76.03%	71.50%	76.25%	72.25%	76.43%	73.50%	80.87%	-	-
3 Hop	64.25%	59.49%	72.25%	77.02%	71.50%	76.64%	73.50%	75.80%	72.00%	76.95%	73.50%	75.80%	74.00%	81.56%	94.25%	94.56%
Others																
1 Hop	66.50%	59.06%	71.71%	71.52%	71.05%	70.67%	73.03%	72.11%	70.39%	69.39%	71.71%	70.75%	88.16%	93.62%	-	-
2 Hop	69.91%	68.87%	72.37%	71.62%	71.71%	72.26%	73.68%	73.68%	69.08%	70.44%	75.00%	75.64%	88.82%	93.99%	-	-
3 Hop	69.00%	72.99%	75.00%	75.00%	71.71%	71.90%	74.34%	73.83%	74.34%	72.73%	73.68%	72.60%	74.34%	74.84%	94.08%	94.67%
Avg.	67.20%	67.84%	71.75%	68.90%	71.39%	68.67%	72.27%	69.28%	71.34%	68.27%	72.45%	69.45%	75.36%	82.26%	94.44%	94.39%
								Purpose								
Functionality																
1 Hop	85.90%	88.89%	89.89%	90.21%	88.83%	89.18%	89.63%	89.82%	89.63%	89.87%	89.89%	89.95%	90.69%	90.57%	-	-
2 Hop	86.97%	87.59%	89.36%	90.05%	88.83%	89.34%	91.76%	91.91%	88.30%	89.00%	92.02%	92.19%	92.55%	92.55%	-	-
3 Hop	85.64%	86.43%	90.43%	90.77%	91.22%	91.52%	90.69%	90.86%	90.69%	90.96%	92.29%	92.54%	92.29%	92.39%	96.81%	96.83%
Advertisement																
1 Hop	70.24%	73.96%	84.52%	80.88%	85.71%	82.61%	86.90%	84.29%	84.52%	80.88%	86.31%	83.21%	83.33%	90.60%	-	-
2 Hop	78.57%	80%	85%	80.60%	82.74%	77.17%	84.52%	80.60%	83.33%	78.46%	83.93%	79.39%	82.74%	90.24%	-	-
3 Hop	79.76%	80.46%	85.71%	82.09%	83.93%	79.07%	86.90%	84.29%	85.71%	82.35%	88.10%	85.51%	82.14%	87.70%	97.02%	96.55%
Analytics																
1 Hop	61.98%	70.45%	70.31%	70.16%	67.71%	65.56%	68.23%	68.06%	70.83%	71.13%	69.71%	70.10%	76.04%	86.39%	-	-
2 Hop	70.83%	75.65%	67.71%	69.31%	69.79%	70.10%	67.19%	68.66%	67.71%	72.57%	72.92%	74.51%	76.04%	86.39%	-	-
3 Hop	70.83%	74.07%	70.31%	71.36%	71.35%	74.42%	70.31%	71.07%	69.79%	69.79%	70.31%	70.16%	76.04%	86.39%	96.81%	96.83%
Others																
1 Hop	72.50%	73.81%	71.25%	73.56%	72.50%	73.17%	75.00%	76.19%	67.50%	68.29%	71.25%	72.29%	90.00%	94.74%	-	-
2 Hop	68.80%	69.14%	76.25%	75.95%	77.50%	77.50%	77.50%	76.92%	76.25%	75.95%	75.00%	75.00%	91.25%	95.30%	-	-
3 Hop	72.50%	73.17%	73.75%	74.07%	75.00%	76.19%	77.50%	78.05%	76.25%	77.11%	77.50%	78.05%	90.00%	94.74%	98.75%	98.59%
Avg.	75.38%	77.80%	79.54%	79.08%	79.59%	78.82%	80.51%	80.06%	79.21%	78.86%	80.77%	80.24%	85.26%	90.67%	97.34%	97.20%

is often implemented in the second and third hops [25]. Since we use 100 paths to represent 2 or 3 hops, it can result in an incomplete representation which may not capture relevant paths to identify `Sharing`.

Overall, we found an increase in the performance for most cases; thus, for all subsequent experiments, we tokenize non-terminal nodes (and include attention).

6.1.2. RQ 1.2: LSTM vs. Bi-LSTM Layers. In comparison to *Exp 1:L_100*, in *Exp 4:Bi_100*, we did not observe an increase in the scores when we used Bi-LSTM layers (see Table 2). The average accuracy for *Practice* and *Purpose* labels are 71.34% and 79.21% which are similar to those for LSTM layers (71.75% and 79.54%, *Exp 1:L_100*). In both experiments 1 and 4, we used 100 AST paths. These results suggest that an additional backward encoding of AST paths is not very significant, especially when the number of paths is small (i.e., 100).

However, with a larger number of AST paths, say 200 or 300, Bi-LTSM could potentially improve the scores. We inspect this assumption in RQ 1.3 when we experiment with both LSTM and Bi-LSTM models.

6.1.3. RQ 1.3: Number of AST Paths. Overall, we find that with a smaller number of paths (100), LSTM predicts labels with higher accuracy while with a larger number of paths (300), Bi-LSTM performs better. When we use the LSTM layer (i.e., *Exp 1:L_100*, *Exp 2:L_200*, and *Exp 3:L_300*), we notice minimal changes in performance between 100, 200, and 300 AST paths as shown in Table 2. The average accuracy for *Practice* labels are 71.75%, 71.39%, and 72.27% and for *Purpose* are 79.54%, 79.59%, and 80.51% across these experiments. These scores differ within the range of ∼1% which can be attributed to the number of actual paths in a code sample.

We computed the average number of AST paths in each individual hop and found that the first hop consists of 80 paths, and the second and third hops consist of ∼100 paths each. Thus, when we extract, say, 300 paths from a 1_Hop code sample, more than half of the paths are null paths (i.e., only padding). However, when we extract the same number of paths from a 3_Hop code sample, where most paths are not null, we observe a slight increase in the scores. For example, in *Exp 3:L_300*, where we extract 300 paths, `Collecting_3_Hop` has an accuracy of 75.66% whereas `Collecting_1_Hop` has an accuracy of 72.37% which matches with our assumption regarding the percentage of null paths in each dataset.

On the other hand, for Bi-LSTM layers, we notice that with a higher number of paths, there is an overall increase in scores. The average accuracy scores for *Practice* are 71.34%, 72.45%, and 75.36% and for *Purpose* are 79.21%, 80.77%, and 85.26%, as shown in Table 2 (i.e., *Exp 4:Bi_100*, *Exp 5:Bi_200*, and *Exp 6:Bi_300*). As noted in Section 6.1.2, with only 100 AST paths, the backward encoding of AST paths provides an insignificant increase to the classification scores. However, as we increase the number of paths from 100 to 200 and 300, we observe that this additional encoding noticeably improves the scores. This is also evident based on the increasing differences in accuracy between LSTM and Bi-LSTM layers. For example, the difference between LSTM and Bi-LSTM layers when using 100 AST paths (i.e., *Exp 1:L_100* vs. *Exp 4:Bi_100*) for *Practice* labels is less than 1%. However, when we increase the number of paths to 300, this difference is 3% with a much larger F-1 score increase of 12% (i.e., *Exp 3:L_300* vs. *Exp 6:Bi_300*). We observe a similar but more conspicuous trend with *Purpose* labels, where the accuracy and F-1 score differences between LSTM and Bi-LSTM layers with 300 paths are ∼5% and ∼10%, respectively. Similar to the LSTM layer's case, using 300 paths for 3_Hop datasets provides better accuracy. These findings suggest that to represent each code sample, 300 paths is the optimal choice, where each hop contributes to 100 paths. Since each hop on average consists of 100 paths, this number makes sense.

To summarize the results from RQ 1.1-1.3, we find that the optimal configurations are to: tokenize non-terminal nodes in AST paths when using an attention-based model; use LSTM layers for fewer paths (100) and Bi-LSTM layers when having a greater number of paths (300); and to represent a sample with 100 paths for each method in the sample. We leverage these findings for our multi-head encoder model with attention, where we tokenize each non-terminal node in an AST path. In each encoder head, we use LSTM layers, because each head is provided with 100 paths for a single hop.

6.2. RQ 2: Classification Accuracy

The *Multi-Head Encoder* column of Table 2 shows that our model classifies labels with significantly better accuracy than other models. The confusion matrices in Appendix B - Figures 9 and 10 also demonstrate the unbiased performance of our model for each label.

We first compare the quantitative scores of our multi-head encoder model with the closest model configuration (*Exp 3:L_300*). Both models use LSTM layers and an attention module. In Experiment 3, we used 300 AST paths to represent each code sample which is the same number of paths used for the multi-head encoder model (100 AST paths from each hop). The key difference between the two models is their architecture. The average accuracy scores for *Practice* and *Purpose* categories in *Exp 3:L_300* are 72.27% and 80.51%, which are ∼22% and ∼17% lower than that of *Multi-Head Encoder*. This improvement in accuracy is also evident when we compare scores with the *Baseline*, where the increase in average scores for *Practice* and *Purpose* are ∼27% and ∼22%, respectively.

These significant improvements in the scores can be attributed to the architectural aspects of the model and the optimal configurations. The three encoder heads encode each hop separately, creating better representations and semantically separating them. This is evident from the results for `Sharing`, for which our model provides a ∼20% increase in accuracy than the closest comparison model (*Exp 3:L_300*). This is because we use 100 AST paths each to represent second and third hops, which ensures that we capture paths that implement `'Sharing'`, such as calls to third-party libraries. This was not possible in the baseline approach as noted in RQ 1.1. Furthermore, in each head, the attention module needs to attend over only 100 AST paths which makes it easier for the module to focus on identifiers (which embed the calls to third-party libraries). This is despite the tokenization of non-terminal nodes, which we noted was the probable cause of a decrease in the accuracy for predicting Sharing label in RQ 1.1. Lastly, each head in the multi-head encoder model uses LSTM layers which effectively encode 100 AST paths. Contrarily, in the closest comparison model, the LSTM layers are required to encode 300 paths which is not an optimal configuration. Even if we switch layers in the comparison model to Bi-LSTM, which encodes 300 paths (*Exp 6:Bi_300*), the results are not nearly as good as the multi-head encoder model.

We also observe that our multi-head encoder model accurately predicts the negative label, as demonstrated by the near diagonal matrices shown in Appendix B, indicating that it predicts without bias towards the positive class. Similarly, our model provides $\sim$30% improvement with Analytics label which achieves low classification scores for every other configuration from *Exp 1:L_100* - *Exp 6:Bi_300*. Similar to Sharing, Analytics samples also have third-party libraries calling permission-requiring APIs to access personal information, often in the second and third hops. To identify this label, identifier information is necessary. We find that these code samples are often obfuscated, primarily to preserve their methodology for analytics. This obfuscation makes it especially challenging for other models to predict this label. However, with individual attention for each hop, attending over 100 paths, it becomes easier to identify third-party library calls to permission-requiring APIs.

To summarize, representing each hop as an individual method, preserves semantic differences between three hops and results in significant improvements in the accuracy of the multi-head encoder model. The classification scores of this model are higher than any other model configuration (i.e. *Exp 1:L_100* - *Exp 6:Bi_300*).

6.3. RQ 3: Localization Efficacy

Our results show that while there is room for improvement, fine-grained localization is efficacious; that is, it helps in writing privacy statements and can accurately identify statements that implement privacy behaviors.

6.3.1. Initial Analysis. We first examine if our automated approach identifies statements that implement privacy behaviors. Our manual evaluation indicates that our approach succeeds in identifying privacy statements. For instance, consider Figures 4 and 5 which show an enumerated and highlighted code sample and its AST paths with the highest attention. The code sample gets the location of the user in the first hop (Figure 4 (a)), calculates its distance to another point (in the second hop (Figure 4 (b)), and then shows this distance, which is to a real-estate property, in the third hop (Figure 4 (c)). When we look at the localized statements in the first hop, we observe that most attention is given to statements that get the user's location (i.e., statements 1, 4, and 5 in Figure 4 (a)), which is the core logic of the first hop and it is implemented in those highlighted statements. These statements are localized based on the terminal nodes of AST paths 1, 4, and 5 shown in Figure 5 (a). Our script does not map AST path 6 shown in the same Figure, since it is obfuscated. Based on the non-terminal nodes 'IfStatement' and 'ReturnStatement', we can approximate the location to lines 8 or 15. However, it cannot be localized with reasonable certainty; hence, we do not map this path.

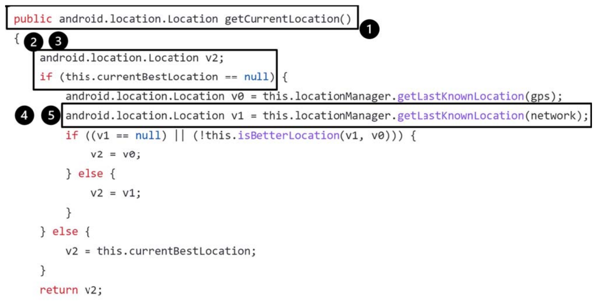
For the second hop, focus is given to statements that get the location and calculate its distance to the user's location (i.e., statements 4 and 5 in Figure 4 (b)). Similar to the first hop, the terminal nodes in AST paths 4 and 5 (Figure 5 (b)) are used to map them. For path 4, the two 'MethodInvocation' non-terminal nodes are also used to verify the mapping. This example shows the efficacy of tokenizing non-terminal nodes. Note that in statement 4, the first hop 'getCurrentLocation' is highlighted which links the first and second hops together.

Lastly, in the third hop, most statements focus on developing a view to show details of the property (i.e., statements 1, 2, and 4). In statement 4, the second hop 'getDistancetoPlace' is also called, which links the second and third hops together. Method names summarize a method's behavior [2], [25] and can be helpful in the comprehension of privacy behaviors. However, this may not always be the case. Consider the method names of the first, second, and third hop in Figure 4. The method names in the first two hops are relevant and provide context about how location is being used, i.e., getting the current location and the distance to the current location which are both highlighted in the source code. But, the method name for the third hop does not provide any insight regarding its privacy behavior, which is, thus, not localized in this sample. This suggests that our approach can detect when method names can be helpful and when they cannot (i.e., not localize them). Statements 1, 2, and 4 are spread out, but our approach precisely identifies them without selecting other statements in the same block that are irrelevant. This supports our design decision of statement-level localization.

Overall, these results indicate that using attention, we can *localize* statements that implement privacy behaviors across three hops as well as the calls to previous hops which is helpful for tracing the flow of information, especially in larger code segments. These results suggest that by mapping highly weighted AST paths, we can provide fine-grained localization of privacy behaviors. Figures 12 and 13 in Appendix D show another example of AST paths to source code mappings.

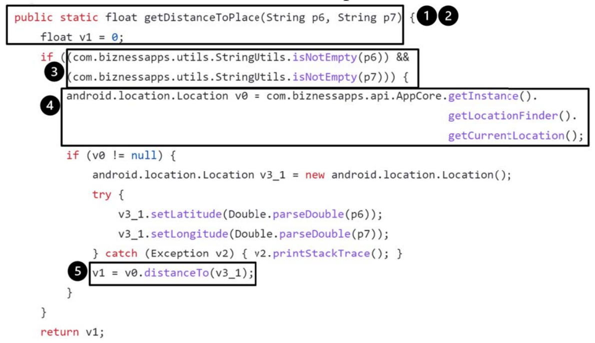
6.3.2. RQ 3.1: Analyzing Privacy Statements. We qualitatively evaluate the dis/advantages of fine-grained localization in helping software professionals write privacy statements. As stated in Section 5.2, we ask six software professionals (i.e., annotators) that are divided into two groups to write simple privacy statements for 20 code samples. These two groups are given the same samples, however, the samples that are localized for one group are not localized for the other group and vice-versa. The results of this study indicate that fine-grained localization helps annotators easily identify relevant code statements that are necessary for writing privacy statements and saves them the effort and time required to read every line of code to understand the privacy behaviors of the code. The localized statements help annotators with comparatively less experience in software development or privacy to create privacy statements and save up to 74% of time.

For each annotator, we discovered negligible differences between the quality of privacy statements for lo-



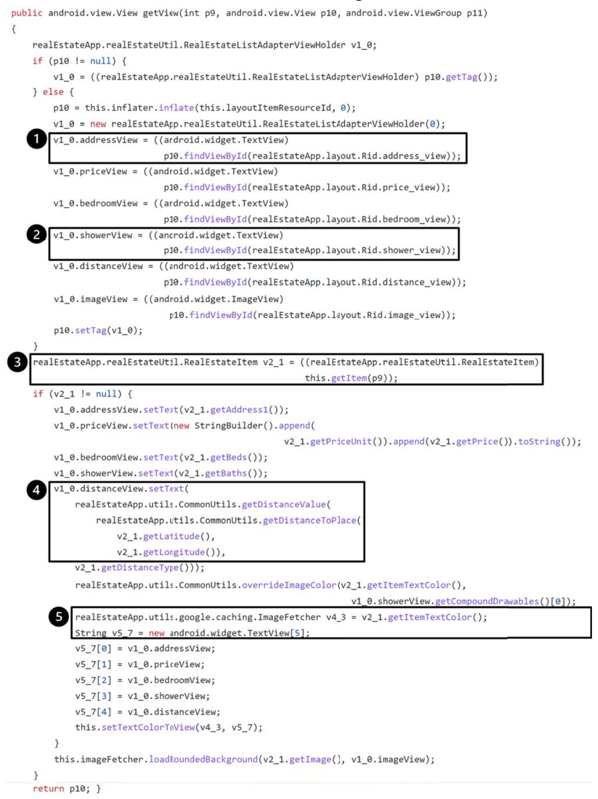
```
public android.location.Location getCurrentLocation() ❶
{ ❷ ❸
    android.location.Location v2;
    if (this.currentBestLocation == null) {
        android.location.Location v0 = this.locationManager.getLastKnownLocation(gps);
❹ ❺     android.location.Location v1 = this.locationManager.getLastKnownLocation(network);
        if ((v1 == null) || (!this.isBetterLocation(v1, v0))) {
            v2 = v0;
        } else {
            v2 = v1;
        }
    } else {
        v2 = this.currentBestLocation;
    }
    return v2;
```

(a) First hop



```
public static float getDistanceToPlace(String p6, String p7) { ❶ ❷
    float v1 = 0;
    if ((com.biznessapps.utils.StringUtils.isNotEmpty(p6)) &&
 ❸      (com.biznessapps.utils.StringUtils.isNotEmpty(p7))) {
 ❹      android.location.Location v0 = com.biznessapps.api.AppCore.getInstance().
                                                               getLocationFinder().
                                                               getCurrentLocation();
        if (v0 != null) {
            android.location.Location v3_1 = new android.location.Location();
            try {
                v3_1.setLatitude(Double.parseDouble(p6));
                v3_1.setLongitude(Double.parseDouble(p7));
            } catch (Exception v2) { v2.printStackTrace(); }
 ❺          v1 = v0.distanceTo(v3_1);
        }
    }
    return v1;
```

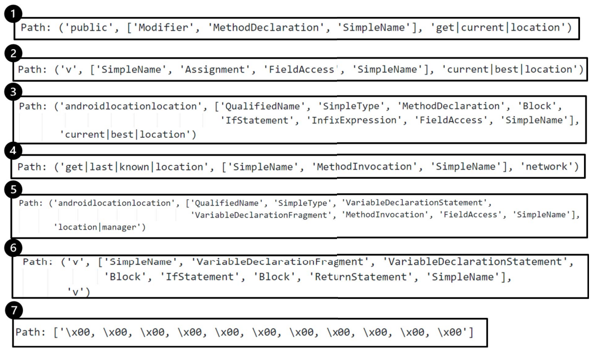
(b) Second hop



```
public android.view.View getView(int p9, android.view.View p10, android.view.ViewGroup p11)
{
    realEstateApp.realEstateUtil.RealEstateListAdapterViewHolder v1_0;
    if (p10 != null) {
        v1_0 = ((realEstateApp.realEstateUtil.RealEstateListAdapterViewHolder) p10.getTag());
    } else {
        p10 = this.inflater.inflate(this.layoutItemResourceId, 0);
        v1_0 = new realEstateApp.realEstateUtil.RealEstateListAdapterViewHolder(0);
 ❶      v1_0.addressView = ((android.widget.TextView)
                            p10.findViewById(realEstateApp.layout.Rid.address_view));
        v1_0.priceView = ((android.widget.TextView)
                            p10.findViewById(realEstateApp.layout.Rid.price_view));
        v1_0.bedroomView = ((android.widget.TextView)
                            p10.findViewById(realEstateApp.layout.Rid.bedroom_view));
 ❷      v1_0.showerView = ((android.widget.TextView)
                            p10.findViewById(realEstateApp.layout.Rid.shower_view));
        v1_0.distanceView = ((android.widget.TextView)
                            p10.findViewById(realEstateApp.layout.Rid.distance_view));
        v1_0.imageView = ((android.widget.ImageView)
                             p10.findViewById(realEstateApp.layout.Rid.image_view));
        p10.setTag(v1_0);
    }
❸  realEstateApp.realEstateUtil.RealEstateItem v2_1 = ((realEstateApp.realEstateUtil.RealEstateItem)
                                                   this.getItem(p9));
    if (v2_1 != null) {
        v1_0.addressView.setText(v2_1.getAddress1());
        v1_0.priceView.setText(new StringBuilder().append(
                                              v2_1.getPriceUnit()).append(v2_1.getPrice()).toString());
        v1_0.bedroomView.setText(v2_1.getBeds());
        v1_0.showerView.setText(v2_1.getBaths());
 ❹      v1_0.distanceView.setText(
            realEstateApp.utils.CommonUtils.getDistanceValue(
                realEstateApp.utils.CommonUtils.getDistanceToPlace(
                    v2_1.getLatitude(),
                    v2_1.getLongitude()),
            v2_1.getDistanceType()));
            realEstateApp.utils.CommonUtils.overrideImageColor(v2_1.getItemTextColor(),
                                                  v1_0.showerView.getCompoundDrawables()[0]);
    ❺      realEstateApp.utils.google.caching.ImageFetcher v4_3 = v2_1.getItemTextColor();
            String v5_7 = new android.widget.TextView[5];
            v5_7[0] = v1_0.addressView;
            v5_7[1] = v1_0.priceView;
            v5_7[2] = v1_0.bedroomView;
            v5_7[3] = v1_0.showerView;
            v5_7[4] = v1_0.distanceView;
            this.setTextColorToView(v4_3, v5_7);
        }
        this.imageFetcher.loadRoundedBackground(v2_1.getImage(), v1_0.imageView);
    }
    return p10; }
```

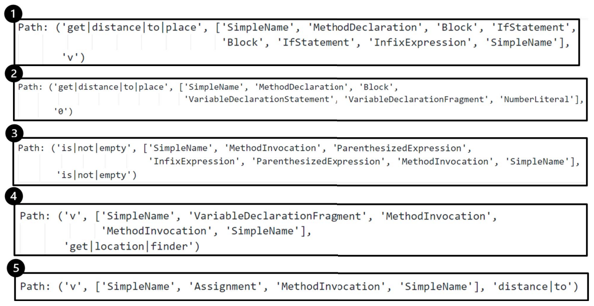
(c) Third hop

Figure 4: Code snippets and localized statements.



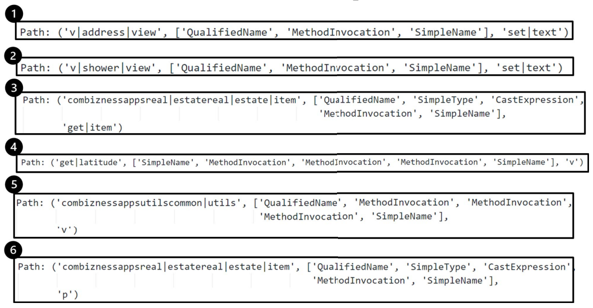
```
❶ Path: ('public', ['Modifier', 'MethodDeclaration', 'SimpleName'], 'get|current|location')
❷ Path: ('v', ['SimpleName', 'Assignment', 'FieldAccess', 'SimpleName'], 'current|best|location')
❸ Path: ('androidlocationlocation', ['QualifiedName', 'SinpleType', 'MethodDeclaration', 'Block',
                                   'IfStatement', 'InfixExpression', 'FieldAccess', 'SimpleName'],
        'current|best|location')
❹ Path: ('get|last|known|location', ['SimpleName', 'MethodInvocation', 'SimpleName'], 'network')
❺ Path: ('androidlocationlocation', ['QualifiedName', 'SimpleType', 'VariableDeclarationStatement',
                                   'VariableDeclarationFragment', 'MethodInvocation', 'FieldAccess', 'SimpleName'],
        'location|manager')
❻ Path: ('v', ['SimpleName', 'VariableDeclarationFragment', 'VariableDeclarationStatement',
              'Block', 'IfStatement', 'Block', 'ReturnStatement', 'SimpleName'],
        'v')
❼ Path: ['\x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00']
```

(a) First hop



```
❶ Path: ('get|distance|to|place', ['SimpleName', 'MethodDeclaration', 'Block', 'IfStatement',
                                  'Block', 'IfStatement', 'InfixExpression', 'SimpleName'],
        'v')
❷ Path: ('get|distance|to|place', ['SimpleName', 'MethodDeclaration', 'Block',
                                  'VariableDeclarationStatement', 'VariableDeclarationFragment', 'NumberLiteral'],
        '0')
❸ Path: ('is|not|empty', ['SimpleName', 'MethodInvocation', 'ParenthesizedExpression',
                         'InfixExpression', 'ParenthesizedExpression', 'MethodInvocation', 'SimpleName'],
        'is|not|empty')
❹ Path: ('v', ['SimpleName', 'VariableDeclarationFragment', 'MethodInvocation',
              'MethodInvocation', 'SimpleName'],
        'get|location|finder')
❺ Path: ('v', ['SimpleName', 'Assignment', 'MethodInvocation', 'SimpleName'], 'distance|to')
```

(b) Second hop



```
❶ Path: ('v|address|view', ['QualifiedName', 'MethodInvocation', 'SimpleName'], 'set|text')
❷ Path: ('v|shower|view', ['QualifiedName', 'MethodInvocation', 'SimpleName'], 'set|text')
❸ Path: ('combiznessappsreal|estatereal|estate|item', ['QualifiedName', 'SimpleType', 'CastExpression',
                                                     'MethodInvocation', 'SimpleName'],
        'get|item')
❹ Path: ('get|latitude', ['SimpleName', 'MethodInvocation', 'MethodInvocation', 'MethodInvocation', 'SimpleName'], 'v')
❺ Path: ('combiznessappsutilscommon|utils', ['QualifiedName', 'MethodInvocation', 'MethodInvocation',
                                           'MethodInvocation', 'SimpleName'],
        'v')
❻ Path: ('combiznessappsreal|estatereal|estate|item', ['QualifiedName', 'SimpleType', 'CastExpression',
                                                     'MethodInvocation', 'SimpleName'],
        'p')
```

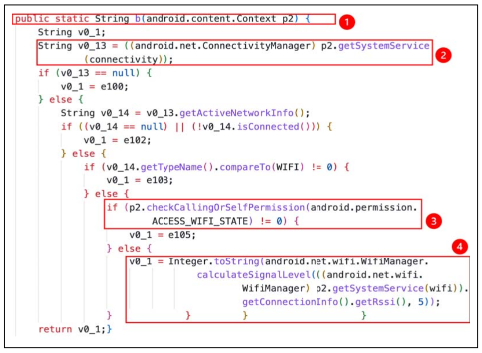
(c) Third hop

Figure 5: Most attended AST paths in each hop.

calized and non-localized samples of similar lengths. For example, Annotator #1 wrote "We collect device information such as MAC address, Model, OS version, and Serial number." for a localized sample and "We check if your device is connected to a network before loading events." for a non-localized one. Both of these statements are written with equal accuracy and detail about the privacy behavior of the code, regardless of fine-grained localization. This level of similarity between statements can be attributed to the annotators' knowledge of privacy, which helps them identify privacy behaviors of the code samples even without localization highlights.

We also compared the time taken by annotators to write privacy statements and discovered that while localization highlights saved time for both groups, Group #2 annotators benefited the most by saving up to 74% of their time. Annotator # 4's average time to write a privacy statement, for instance, dropped from 9.7 minutes to 5.5 minutes with localized samples because the highlighted lines helped them by "drawing [their] attention to privacy-related lines." For Group #1 annotators, we found that localization highlights were more helpful in some scenarios than others. Consider the localization highlights in Figure 6. Annotator #2 noted that localization highlights were most helpful since they accurately highlighted "incidental variable assignments and logic syntax" in a larger code snippet. These highlights reduced the time and effort required for reading and comprehending privacy behaviors. On the other hand, fine-grained localization was less helpful when there were too many or too few highlights. For example, in Appendix E - Figure 14 almost all statements of a small method are highlighted, rendering localization ineffective. Similarly, a single highlight of a larger method in Appendix E - Figure 15 did not reduce the number of lines required to read and understand privacy behaviors.

Lastly, we compared the privacy statements between groups for the same samples and discovered Group #1's statements were either equally good or of better quality compared to those of Group #2, regardless of localization highlights. For example, Annotator #3 wrote "Opens google maps to the devices current location and returns lat/lon when the user clicks" without localization whereas Annotator #5 wrote "We use your location to show your position graphically on Google Maps." with localization for the same sample. While the two statements are comparable, the one written by Annotator #3 has more detail which currently lacks in the statement written by Annotator #5. These differences in quality and detail can be attributed to the differences in annotators' experiences with privacy and software development. These findings also suggest that developers with limited or no knowledge about privacy could write privacy statements that are comparable to those written by privacy experts when they are provided with localized code samples. However, a more rigorous evaluation will be required to confirm this conclusion, which we plan to do in the future.



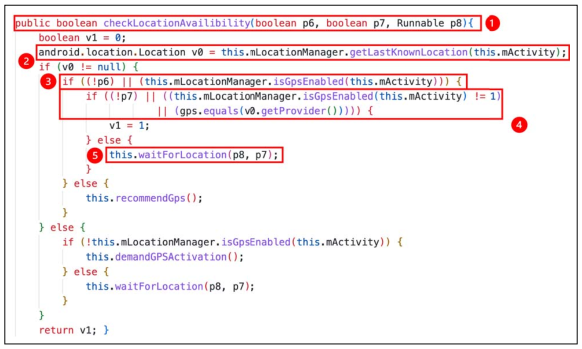
```
public static String b(android.content.Context p2) {
    String v0_1;
    String v0_13 = ((android.net.ConnectivityManager) p2.getSystemService
            (connectivity));
    if (v0_13 == null) {
        v0_1 = e100;
    } else {
        String v0_14 = v0_13.getActiveNetworkInfo();
        if ((v0_14 == null) || (!v0_14.isConnected())) {
            v0_1 = e102;
        } else {
            if (v0_14.getTypeName().compareTo(WIFI) != 0) {
                v0_1 = e103;
            } else {
                if (p2.checkCallingOrSelfPermission(android.permission.
                        ACCESS_WIFI_STATE) != 0) {
                    v0_1 = e105;
                } else {
                    v0_1 = Integer.toString(android.net.wifi.WifiManager.
                            calculateSignalLevel(((android.net.wifi.
                                    WifiManager) p2.getSystemService(wifi)).
                                    getConnectionInfo().getRssi(), 5));
                }       }           }                   }
    return v0_1;}
```

Figure 6: Appropriate highlights for larger code sample was most helpful for all annotators.

6.3.3. RQ 3.2: Localization Accuracy. For quantitative evaluation of localization, as mentioned in Section 5.2, we asked the annotators in Group #1 to label 230 fine-grained localized (highlighted) statements across 20 code samples. These annotators gave a binary label to each localized statement based on whether it implements privacy behavior or not.

Annotator #1 found 121 highlights out of 230 (∼52%) as statements that implement privacy behaviors, while Annotator #2 found 148 highlights (∼65%) and Annotator #3 found 174 statements (∼75%) as relevant. The inter-annotator agreement scores, i.e., Fleiss's Kappa and Krippendorff's Alpha, among all three annotators were 0.362 for both scores, which for Kappa is considered as "Fair Agreement". Although our Kappa value is slightly low, other works [7], [18], [31] have also achieved low agreement scores since even experts often disagree [8], [31]. We computed the percentages of agreements among annotators for a more thorough analysis. We found that in the best case (where at least one annotator responded 'yes'), 85% of highlighted statements were relevant, and in the average case (where the majority of annotators responded 'yes'), 65% of highlighted statements were relevant. This distribution of agreements indicates that all annotators found that the majority of highlighted statements implemented privacy behaviors, where these statements are spread across multiple hops. Their annotations also indicate that our approach has some noise and there are highlighted statements that do not implement privacy behaviors. Thus, there is some scope for improvement in the process of localizing privacy code statements.

We further analyzed the annotation results by inspecting individual cases where annotators disagreed/agreed. We found that, in many cases, annotators agreed when code snippets explicitly used personal information, such as location or email, but disagreed on their subjective view of which statements actually constituted as 'implementing privacy behaviors'. Consider the code snippet in Figure 16, which initializes a `SensorManager` to access the devices' accelerometer. While Annotators #2 and #3 annotated with 'yes' for statements 2,3, and 4, Annotator #1 disagreed. For statement 1, Annotator #3 responded with a 'yes' but Annotators #1 and #2 disagreed since these statements provide *implicit* access to a personal accelerometer (which may not be considered as personal information by some). These demonstrate the subjectivity of privacy that exists even between experts which is reported in literature [8], [31]. Although the code snippet in Figure 16 does not explicitly consume the sensor information, the method name and the parameters used can help understand *how* and *why* the information is used.



```
public boolean checkLocationAvailibility(boolean p6, boolean p7, Runnable p8){
    boolean v1 = 0;
    android.location.Location v0 = this.mLocationManager.getLastKnownLocation(this.mActivity);
    if (v0 != null) {
        if ((!p6) || (this.mLocationManager.isGpsEnabled(this.mActivity))) {
            if ((!p7) || ((this.mLocationManager.isGpsEnabled(this.mActivity) != 1)
                    || (gps.equals(v0.getProvider())))) {
                v1 = 1;
            } else {
                this.waitForLocation(p8, p7);
            }
        } else {
            this.recommendGps();
        }
    } else {
        if (!this.mLocationManager.isGpsEnabled(this.mActivity)) {
            this.demandGPSActivation();
        } else {
            this.waitForLocation(p8, p7);
        }
    }
    return v1; }
```

Figure 7: Statements explicitly using personal information (e.g., 2-5), implement privacy behaviors.

Consider another example in Appendix F - Figure 17, which shows a code snippet from an advertisement library using personal information. Here, all three annotators found the use of 'traditional' personal information as implementing privacy behaviors and completely agreed on the highlights. Specifically, they agreed that highlights 4-6 and 8-11 localize privacy behaviors. This example demonstrates that our approach identifies the use of sensitive personal information that is not captured by API calls. For example, date of birth, gender, income, and ethnicity are highly sensitive personal information that are not provided via API calls. It is to be noted that the code samples in the dataset are *not* annotated with any localization information and therefore, all predictions about privacy-relevant statements are learned by the model on its own. Hence, our approach is able to identify such privacy behaviors and can help developers provide much granular insight into their implementations.

7. Limitations

We plan to address the following limitations in future.
Automated mapping: One challenge of our automated approach is to provide an exact match of every path with a statement in the source code. Since a path is the traversal of a sub-tree, it may contain terminal and non-terminal nodes of different statements, making it difficult to provide a one-to-one mapping. In some cases, code samples and thus, the terminal nodes are obfuscated which makes mapping challenging. Thus, our script sometimes maps one path to several statements (i.e., one-to-many), since it only matches terminal nodes. To mitigate this challenge, we manually verify and correct the automated mappings based on non-terminal nodes (such as using `IfExpression` tokens). If we cannot map the paths with reasonable certainty, we do not map them at all. In the future, we will revise our model architecture to provide source code tokens along with the Abstract Syntax Tree (not paths of AST) which will allow the model to map terminal and non-terminal nodes to source code tokens thereby eliminating the need for separate scripts for mapping. While our approach does not always map obfuscated nodes, developers who will use our work will have access to unobfuscated code.
Detecting privacy statements: Our approach accurately identifies privacy-relevant statements in source code in a majority of cases; however, there are instances, when less relevant paths were attended. For example, in the second hop of Figure 4 (b), the model focuses on statements that check whether input parameters are empty (i.e., statements 3 in the figure). The model also attends null paths (i.e., padding), such as statement 7 in Figure 5 (a)). As with any machine learning approach, there are false positive predictions. In the future, we plan to minimize false positive localization by using our revised model which will utilize the contextual information from source code tokens and syntactic information from the AST for localization. We will also use attention masks to allow model to differentiate between padding and true AST paths.
Evaluation: We evaluated our work with 20 samples which may seem trivial, but the ADPAc dataset is prepared from 15,000 real Android applications [25]. Moreover, we extracted the 20 samples from the test sets of ADPAc dataset which guaranteed that they were not seen during the training phase, thereby evaluating our work on real-world application code. Furthermore, in our qualitative evaluation, we selected annotators with some experience in software development and privacy. However, they are not the authors of the source code they evaluated, nor had access to the entire app's source code which may have impacted accuracy for some samples.

8. Conclusions and Future Work

In this paper, we described a novel approach to provide fine-grained localization of privacy behaviors in an application's source code for generating privacy labels and helping developers write privacy statements. We developed a novel multi-head encoder model that creates individual representations of multiple methods and then uses attention to identify relevant statements in those representations. To identify optimal model configurations, we first conducted six sets of experiments and then trained our model using the optimal configurations. Next, we used our model to predict *Privacy Action* labels. Our quantitative results indicate that our unique architecture significantly outperforms the baseline, and achieves high classification accuracy scores of 91.41% - 98.45% in predicting *Privacy Action* labels. We also evaluated our fine-grained localization approach manually as well as with six software professionals. Manual evaluation results indicate that our automated approach correctly highlights privacy-relevant statements in most cases, but may need manual curation to remove the false positives mappings. We plan to address this challenge in the future as discussed in Section 7.

Our qualitative evaluation demonstrates that our approach helps professionals easily identify relevant code statements that are necessary for writing privacy statements and saves them the effort and time required to read every line of code to understand their privacy behaviors. The time required is reduced up to 74% for professionals with lower expertise. We also evaluated the accuracy of our approach with three of the six professionals with more expertise and found that our model identifies relevant statements that implement privacy behaviors in the majority of the cases. Based on these results, our approach provides a mechanism for fine-grained localization of privacy behaviors. In this work, we only demonstrated the feasibility of fine-grained localization with six software professionals at our university. In the future, we will conduct a user study with several Android application developers to evaluate our localization approach's efficacy, and also perform a comparative analysis on the quality and timing of the privacy statements creation.

Data Availability

Our datasets, models, and training scripts are available on the GitHub page of our project at https://github.com/PERC-Lab/Fine_Grained_Localization

Acknowledgements

We would like to thank Sam Morse, Mac Creamer, Max Prybylo, Adam Green, Sean Radel, Zack Delile, and Theo Brucker for their contributions to this work.

References

[1] App privacy details - app store, 2022. Note accessed 30th September 2022.

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

[3] Kamel Alreshedy, Dhanush Dharmaretnam, Daniel M German, Venkatesh Srinivasan, and T Aaron Gulliver. Scc: automatic classification of code snippets. *arXiv preprint arXiv:1809.07945*, 2018.

[4] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. 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*, pages 23–34, 2017.

[5] Saba Arshad, Munam A Shah, Abdul Wahid, Amjad Mehmood, Houbing Song, and Hongnian Yu. Samadroid: a novel 3-level hybrid malware detection model for android operating system. *IEEE Access*, 6:4321–4339, 2018.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] Rebecca Balebako, Abigail Marsh, Jialiu Lin, Jason I Hong, and Lorrie Faith Cranor. The privacy and security behaviors of smartphone app developers. 2014.

[8] Rebecca Balebako, Richard Shay, and Lorrie Faith Cranor. Is Your Inseam a Biometric? A Case Study on the Role of Usability Studies in Developing Public Policy. In *Proceedings 2014 Workshop on Usable Security*. Internet Society, 2014.

[9] Minghao Chen and Xiaojun Wan. Neural comment generation for source code with auxiliary code classification task. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 522–529. IEEE, 2019.

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

[11] Suzanne Frey. Get more information about your apps in google play, Apr 2022 (access.

[12] Jack Gardner, Yuanyuan Feng, Kayla Reiman, Zhi Lin, Akshath Jain, and Norman Sadeh. Helping Mobile Application Developers Create Accurate Privacy Labels. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 212–230. IEEE, June 2022.

[13] EU GDPR. General Data Protection Regulation. https://gdpr.eu/fines/.

[14] Shlok Gilda. Source code classification using neural networks. In *2017 14th international joint conference on computer science and software engineering (JCSSE)*, pages 1–6. IEEE, 2017.

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

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

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

[18] Hamza Harkous, Sai Teja Peddinti, Rishabh Khandelwal, Animesh Srivastava, and Nina Taft. Hark: A Deep Learning System for Navigating Privacy Feedback at Scale. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2469–2486. IEEE, May 2022.

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

[20] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2269–2275. AAAI Press, 2018.

[21] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. {SUPOR}: Precise and scalable sensitive user input detection for android apps. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 977–992, 2015.

[22] Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M Chen, and Yogachandran Rahulamathavan. Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, 68:36–46, 2017.

[23] Sarthak Jain and Byron C Wallace. Attention is not explanation. *arXiv preprint arXiv:1902.10186*, 2019.

[24] Vijayanta Jain, Sanonda Datta Gupta, Sepideh Ghanavati, and Sai Teja Peddinti. Prigen: Towards automated translation of android applications' code to privacy captions. pages 142–151, 2021.

[25] Vijayanta Jain, Sanonda Datta Gupta, Sepideh Ghanavati, Sai Teja Peddinti, and Collin McMillan. Pact: Detecting and classifying privacy behavior of android applications. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 104–118, 2022.

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

[27] Patrick Gage Kelley, Joanna Bresee, Lorrie Faith Cranor, and Robert W Reeder. A" nutrition label" for privacy. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, pages 1–12, 2009.

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

[29] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. On locating malicious code in piggybacked android apps. *Journal of Computer Science and Technology*, 32(6):1108–1124, 2017.

[30] Tianshi Li, Elijah B Neundorfer, Yuvraj Agarwal, and Jason I Hong. Honeysuckle: Annotation-guided code generation of in-app privacy notices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(3):1–27, 2021.

[31] Tianshi Li, Kayla Reiman, Yuvraj Agarwal, Lorrie Faith Cranor, and Jason I. Hong. Understanding challenges for developers to create accurate privacy nutrition labels. In *CHI Conference on Human Factors in Computing Systems*, page 1–24, New Orleans LA USA, Apr 2022. ACM.

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

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

[34] Yang Liu. Fine-tune bert for extractive summarization. *arXiv preprint arXiv:1903.10318*, 2019.

[35] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 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)*, pages 287–292, 2017.

[36] Zhuo Ma, Haoran Ge, Zhuzhu Wang, Yang Liu, and Ximeng Liu. Droidetec: Android malware detection and malicious code localization through deep learning. *arXiv preprint arXiv:2002.03594*, 2020.

[37] Sayan Maitra, Bohyun Suh, and Sepideh Ghanavati. Privacy consistency analyzer for android applications. In *2018 IEEE 5th International Workshop on Evolving Security & Privacy Requirements Engineering (ESPRE)*, pages 28–33. IEEE, 2018.

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

[39] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. A multi-view context-aware approach to android malware detection and malicious code localization. *Empirical Software Engineering*, 23(3):1222–1274, 2018.

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

[41] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. {WHYPER}: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 527–542, 2013.

[42] Sai Teja Peddinti, Igor Bilogrevic, Nina Taft, Martin Pelikan, Úlfar Erlingsson, Pauline Anthonysamy, and Giles Hogben. Reducing permission requests in mobile apps. In *Proceedings of the Internet Measurement Conference*, pages 259–266, 2019.

[43] Google Play. Developer content policy, 2022 (accessed June 6th, 2022).

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

[45] Radim Rehurek. Gensim. https://radimrehurek.com/gensim/index.html, 2021 (accessed May 1st, 2021.

[46] Sanae Rosen, Zhiyun Qian, and Z Morely Mao. 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*, pages 221–232, 2013.

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

[48] Haque Sakib, LeClair Alexander, Wu Lingfei, and McMillan Collin. Improved automatic summarization of subroutines via attention to file context. *International Conference on Mining Software Repositories*, 2020.

[49] Sofia Serrano and Noah A Smith. Is attention interpretable? *arXiv preprint arXiv:1906.03731*, 2019.

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

[51] Xiaobing Sun and Wei Lu. Understanding attention for text classification. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3418–3428, 2020.

[52] Suman R Tiwari and Ravi U Shukla. An android malware detection technique using optimized permission and api with pca. In *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, pages 2611–2616. IEEE, 2018.

[53] Juriaan Kennedy Van Dam and Vadim Zaytsev. Software language identification with natural language classifiers. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 624–628. IEEE, 2016.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[55] Qing Wu, Peng Sun, Xueshu Hong, Xueling Zhu, and Bo Liu. An android malware detection and malicious code location method based on graph neural network. In *2021 The 4th International Conference on Machine Learning and Machine Intelligence*, pages 50–56, 2021.

[56] Yue Xiao, Zhengyi Li, Yue Qin, Jiale Guan, Xiaolong Bai, Xiaojing Liao, and Luyi Xing. Lalaine: Measuring and characterizing non-compliance of apple privacy labels at scale. *arXiv preprint arXiv:2206.06274*, 2022.

[57] Wei Yang, Yuqing Xie, Aileen Lin, Xingyu Li, Luchen Tan, Kun Xiong, Ming Li, and Jimmy Lin. End-to-end open-domain question answering with bertserini. *arXiv preprint arXiv:1902.01718*, 2019.

[58] Masaharu Yoshioka, Yasuhiro Aoki, and Youta Suzuki. Bert-based ensemble methods with data augmentation for legal textual entailment in coliee statute law task. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Law*, pages 278–284, 2021.

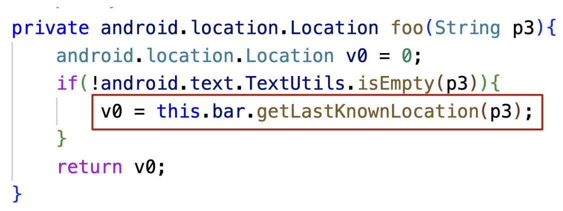
[59] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. Toward automatically generating privacy policy for android apps. *IEEE Transactions on Information Forensics and Security*, 12(4):865–880, 2016.

[60] Sebastian Zimmeck, Rafael Goldstein, and David Baraka. Privacyflash pro: Automating privacy policy generation for mobile apps. In *NDSS*, 2021.

[61] Sebastian Zimmeck, Peter Story, Daniel Smullen, Abhilasha Ravichander, Ziqi Wang, Joel Reidenberg, N Cameron Russell, and Norman Sadeh. Maps: Scaling privacy compliance analysis to a million apps. *Proceedings on Privacy Enhancing Technologies*, 2019(3):66–86, 2019.

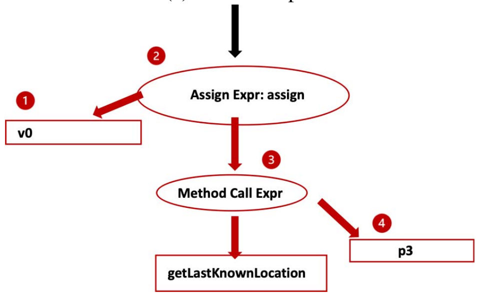
A. Abstract Syntax Tree and AST Paths

Source code can be represented as abstract syntax trees (AST) to show their syntactic structure. Figures 8 (a) and (b) show a code segment and its partial AST. The leaves of an AST, which are identifiers such as variables and names, are called *terminal nodes* (rectangular nodes in Figure 8 (b)). The non-leaves, which represent the syntactic structures such as if-statements and loops, are called *non-terminal nodes* [2] (oval nodes in Figure 8 (b)). Traversing from one terminal node to another is referred to as an AST path. Figure 8 (c) shows a list of AST paths traversed from the partial AST in Figure 8 (b). Since an AST contains useful syntactic information about a code snippet, recent work in code summarization [2], [19], [48] use AST paths to represent code. ADPAc contains the AST paths of code samples and their labels which we use in this work.

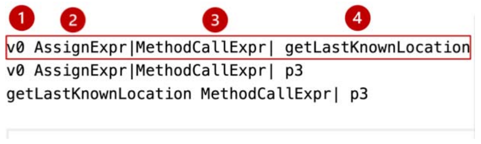


```
private android.location.Location foo(String p3){
    android.location.Location v0 = 0;
    if(!android.text.TextUtils.isEmpty(p3)){
        v0 = this.bar.getLastKnownLocation(p3);
    }
    return v0;
}
```

(a) Code Sample



(b) Abstract Syntax Tree (AST)

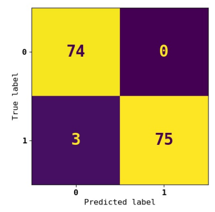


```
v0 AssignExpr|MethodCallExpr| getLastKnownLocation
v0 AssignExpr|MethodCallExpr| p3
getLastKnownLocation MethodCallExpr| p3
```

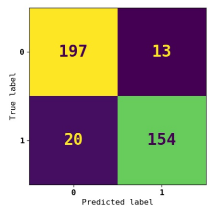
(c) AST Paths

Figure 8: (a) A Code Snippet, (b) Its Partial AST, and (c) Corresponding AST Paths. The Partial AST in (b) Represents the Syntactic Information of the Code Segment Highlighted in Green in (a)

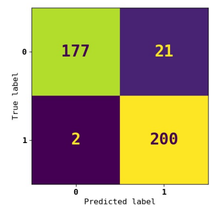
B. RQ 1.2: Confusion Matrices



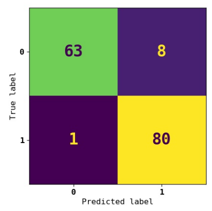
(a) `Collecting`



(b) `Sharing`

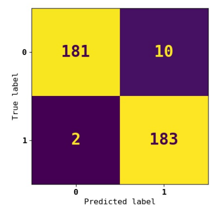


(c) `Processing`

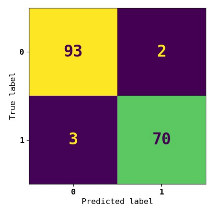


(d) `Others`

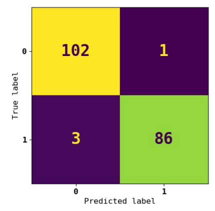
Figure 9: Confusion Matrices for *Purpose* labels. 0 is positive label and 1 is negative label in each dataset. The x-axis shows the predicted label and the y-axis shows the true label.



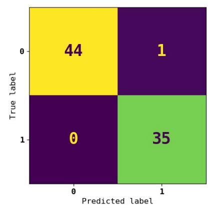
(a) Functionality



(b) Advertisement



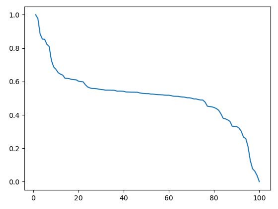
(c) Analytics



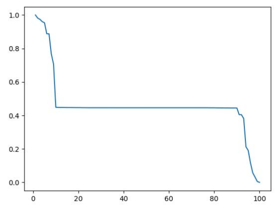
(d) Others

Figure 10: Confusion Matrices for *Purpose* labels. 0 is positive label and 1 is negative label in each dataset. The x-axis shows the predicted label and the y-axis shows the true label.

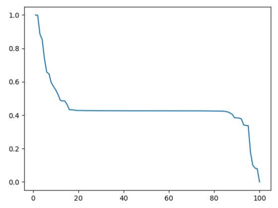
C. RQ 1.3: Attention Maps



(a) First Hop



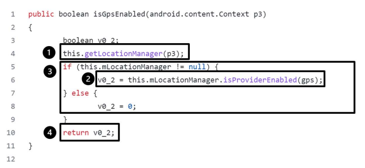
(b) Second Hop



(c) Third Hop

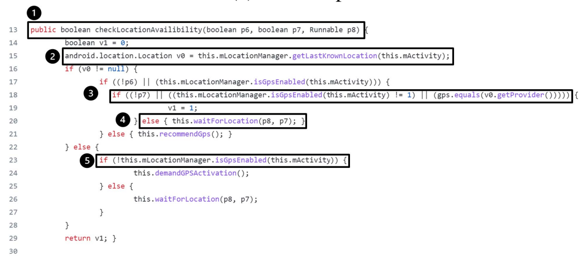
Figure 11: Attention Maps of Individual Hops for Selected Code Sample.

D. RQ 3: Localization Feasibility – Additional Examples



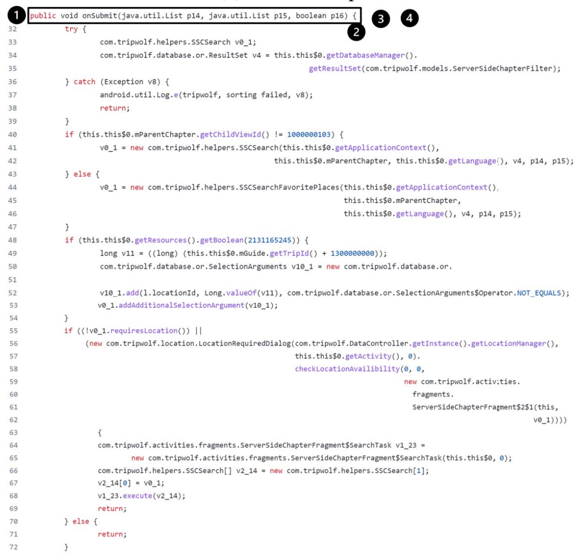
```
public boolean isGpsEnabled(android.content.Context p3)
{
        boolean v0_2;
        this.getLocationManager(p3);
        if (this.mLocationManager != null) {
                v0_2 = this.mLocationManager.isProviderEnabled(gps);
        } else {
                v0_2 = 0;
        }
        return v0_2;
}
```

(a) First Hop



```
public boolean checkLocationAvailibility(boolean p6, boolean p7, Runnable p8) {
        boolean v1 = 0;
        android.location.Location v0 = this.mLocationManager.getLastKnownLocation(this.mActivity);
        if (v0 != null) {
                if ((!p6) || (this.mLocationManager.isGpsEnabled(this.mActivity))) {
                        if ((!p7) || ((this.mLocationManager.isGpsEnabled(this.mActivity) != 1) || (gps.equals(v0.getProvider())))) {
                                v1 = 1;
                        } else { this.waitForLocation(p8, p7); }
                } else { this.recommendGps(); }
        } else {
                if (!this.mLocationManager.isGpsEnabled(this.mActivity)) {
                        this.demandGPSActivation();
                } else {
                        this.waitForLocation(p8, p7);
                }
        }
        return v1; }
```

(b) Second Hop

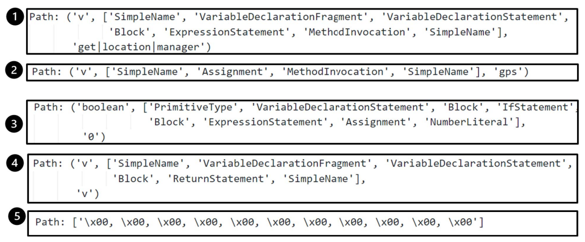


```
public void onSubmit(java.util.List p14, java.util.List p15, boolean p16) {
        try {
                com.tripwolf.helpers.SSCSearch v0_1;
                com.tripwolf.database.or.ResultSet v4 = this.this$0.getDatabaseManager().
                                                        getResultSet(com.tripwolf.models.ServerSideChapterFilter);
        } catch (Exception v8) {
                android.util.Log.e(tripwolf, sorting failed, v8);
                return;
        }
        if (this.this$0.mParentChapter.getChildViewId() != 1000000103) {
                v0_1 = new com.tripwolf.helpers.SSCSearch(this.this$0.getApplicationContext(),
                                                this.this$0.mParentChapter, this.this$0.getLanguage(), v4, p14, p15);
        } else {
                v0_1 = new com.tripwolf.helpers.SSCSearchFavoritePlaces(this.this$0.getApplicationContext(),
                                                        this.this$0.mParentChapter,
                                                        this.this$0.getLanguage(), v4, p14, p15);
        }
        if (this.this$0.getResources().getBoolean(2131165245)) {
                long v11 = ((long) (this.this$0.mGuide.getTripId() + 1300000000));
                com.tripwolf.database.or.SelectionArguments v10_1 = new com.tripwolf.database.or.

                v10_1.add(l.locationId, Long.valueOf(v11), com.tripwolf.database.or.SelectionArguments$Operator.NOT_EQUALS);
               v0_1.addAdditionalSelectionArgument(v10_1);
        }
        if ((!v0_1.requiresLocation()) ||
            (new com.tripwolf.location.LocationRequiredDialog(com.tripwolf.DataController.getInstance().getLocationManager(),
                                                this.this$0.getActivity(), 0).
                                                checkLocationAvailibility(0, 0,
                                                                new com.tripwolf.activities.
                                                                  fragments.
                                                                  ServerSideChapterFragment$2$1(this,
                                                                                        v0_1))))
               {
               com.tripwolf.activities.fragments.ServerSideChapterFragment$SearchTask v1_23 =
                       new com.tripwolf.activities.fragments.ServerSideChapterFragment$SearchTask(this.this$0, 0);
               com.tripwolf.helpers.SSCSearch[] v2_14 = new com.tripwolf.helpers.SSCSearch[1];
               v2_14[0] = v0_1;
               v1_23.execute(v2_14);
               return;
        } else {
               return;
        }
```

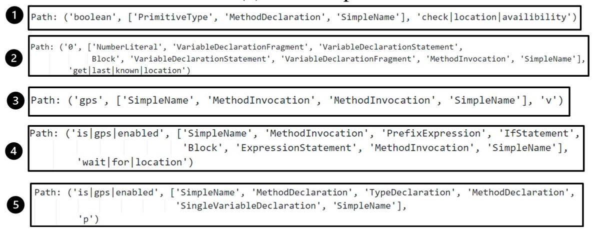
(c) Third Hop

Figure 12: Code Snippets and Localized Statements of Selected Code Sample.



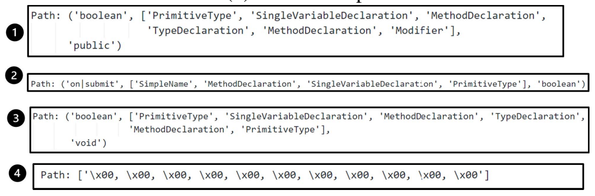
1
```
Path: ('v', ['SimpleName', 'VariableDeclarationFragment', 'VariableDeclarationStatement',
                 'Block', 'ExpressionStatement', 'MethcdInvocation', 'SimpleName'],
        'get|location|manager')
```
2
```
Path: ('v', ['SimpleName', 'Assignment', 'MethodInvocation', 'SimpleName'], 'gps')
```
3
```
Path: ('boolean', ['PrimitiveType', 'VariableDeclarationStatement', 'Block', 'IfStatement',
                      'Block', 'ExpressionStatement', 'Assignment', 'NumberLiteral'],
        '0')
```
4
```
Path: ('v', ['SimpleName', 'VariableDeclarationFragment', 'VariableDeclarationStatement',
                 'Block', 'ReturnStatement', 'SimpleName'],
        'v')
```
5
```
Path: ['\x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00']
```

(a) First Hop



1
```
Path: ('boolean', ['PrimitiveType', 'MethodDeclaration', 'SimpleName'], 'check|location|availibility')
```
2
```
Path: ('0', ['NumberLiteral', 'VariableDeclarationFragment', 'VariableDeclarationStatement',
            Block', 'VariableDeclarationStatement', 'VariableDeclarationFragment', 'MethodInvocation', 'SimpleName'],
        'get|last|known|location')
```
3
```
Path: ('gps', ['SimpleName', 'MethodInvocation', 'MethodInvocation', 'SimpleName'], 'v')
```
4
```
Path: ('is|gps|enabled', ['SimpleName', 'MethodInvocation', 'PrefixExpression', 'IfStatement',
                              'Block', 'ExpressionStatement', 'MethodInvocation', 'SimpleName'],
        'wait|for|location')
```
5
```
Path: ('is|gps|enabled', ['SimpleName', 'MethodDeclaration', 'TypeDeclaration', 'MethodDeclaration',
                             'SingleVariableDeclaration', 'SimpleName'],
        'p')
```

(b) Second Hop

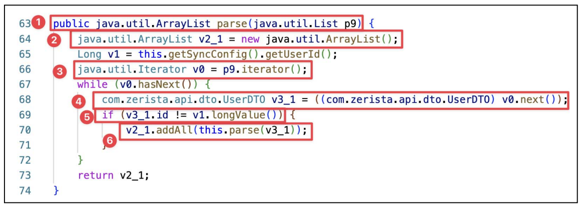


1
```
Path: ('boolean', ['PrimitiveType', 'SingleVariableDeclaration', 'MethodDeclaration',
                      'TypeDeclaration', 'MethodDeclaration', 'Modifier'],
       'public')
```
2
```
Path: ('on|submit', ['SimpleName', 'MethodDeclaration', 'SingleVariableDeclaration', 'PrimitiveType'], 'boolean')
```
3
```
Path: ('boolean', ['PrimitiveType', 'SingleVariableDeclaration', 'MethodDeclaration', 'TypeDeclaration',
                    'MethodDeclaration', 'PrimitiveType'],
       'void')
```
4
```
Path: ['\x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00, \x00']
```

(c) Third Hop

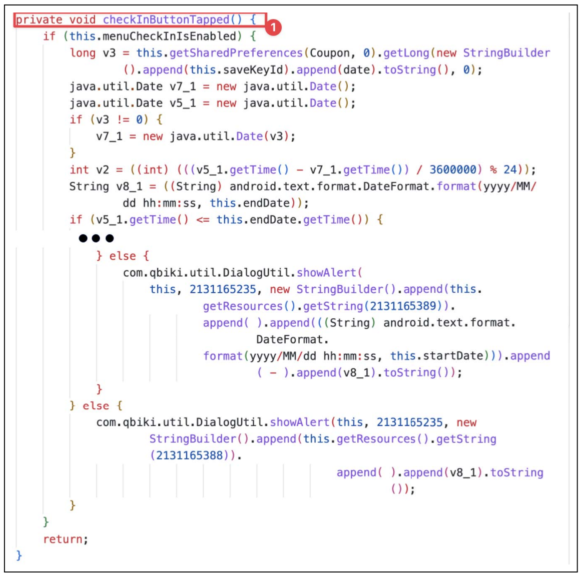
Figure 13: Most Attended AST Paths in Each Hop for Selected Code Sample.

E. RQ 3.1: Analyzing Privacy Statements – Examples



```
public java.util.ArrayList parse(java.util.List p9) {
    java.util.ArrayList v2_1 = new java.util.ArrayList();
    Long v1 = this.getSyncConfig().getUserId();
    java.util.Iterator v0 = p9.iterator();
    while (v0.hasNext()) {
        com.zerista.api.dto.UserDTO v3_1 = ((com.zerista.api.dto.UserDTO) v0.next());
        if (v3_1.id != v1.longValue()) {
            v2_1.addAll(this.parse(v3_1));
        }
    }
    return v2_1;
}
```

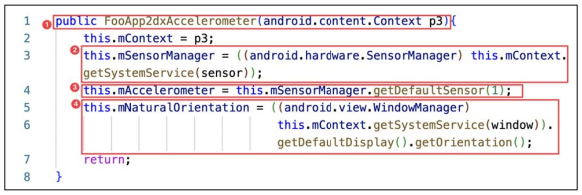
Figure 14: Too many highlights for small code samples rendered localization ineffective.



```
private void checkInButtonTapped() {
    if (this.menuCheckInIsEnabled) {
        long v3 = this.getSharedPreferences(Coupon, 0).getLong(new StringBuilder
                ().append(this.saveKeyId).append(date).toString(), 0);
        java.util.Date v7_1 = new java.util.Date();
        java.util.Date v5_1 = new java.util.Date();
        if (v3 != 0) {
            v7_1 = new java.util.Date(v3);
        }
        int v2 = ((int) (((v5_1.getTime() - v7_1.getTime()) / 3600000) % 24));
        String v8_1 = ((String) android.text.format.DateFormat.format(yyyy/MM/
                dd hh:mm:ss, this.endDate));
        if (v5_1.getTime() <= this.endDate.getTime()) {
         ●●●
            } else {
                com.qbiki.util.DialogUtil.showAlert(
                    this, 2131165235, new StringBuilder().append(this.
                            getResources().getString(2131165389)).
                            append( ).append(((String) android.text.format.
                                    DateFormat.
                            format(yyyy/MM/dd hh:mm:ss, this.startDate))).append
                                    ( - ).append(v8_1).toString());
            }
        } else {
            com.qbiki.util.DialogUtil.showAlert(this, 2131165235, new
                    StringBuilder().append(this.getResources().getString
                    (2131165388)).
                                            append( ).append(v8_1).toString
                                                    ());
        }
    }
    return;
}
```

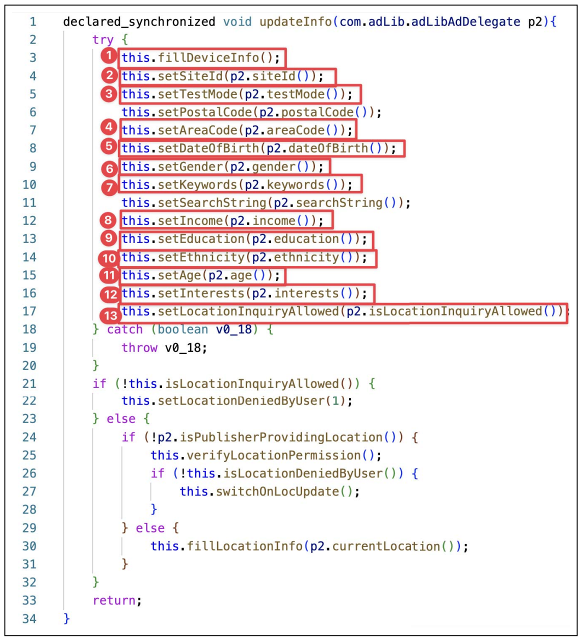
Figure 15: Too few highlights for large code samples do not help.

F. RQ 3.2: Accuracy of Localization – Examples



```
public FooApp2dxAccelerometer(android.content.Context p3){
    this.mContext = p3;
    this.mSensorManager = ((android.hardware.SensorManager) this.mContext.
    getSystemService(sensor));
    this.mAccelerometer = this.mSensorManager.getDefaultSensor(1);
    this.mNaturalOrientation = ((android.view.WindowManager)
                                this.mContext.getSystemService(window)).
                                getDefaultDisplay().getOrientation();
    return;
}
```

Figure 16: Disagreement analysis: code snippet 1



```
declared_synchronized void updateInfo(com.adLib.adLibAdDelegate p2){
    try {
        this.fillDeviceInfo();
        this.setSiteId(p2.siteId());
        this.setTestMode(p2.testMode());
        this.setPostalCode(p2.postalCode());
        this.setAreaCode(p2.areaCode());
        this.setDateOfBirth(p2.dateOfBirth());
        this.setGender(p2.gender());
        this.setKeywords(p2.keywords());
        this.setSearchString(p2.searchString());
        this.setIncome(p2.income());
        this.setEducation(p2.education());
        this.setEthnicity(p2.ethnicity());
        this.setAge(p2.age());
        this.setInterests(p2.interests());
        this.setLocationInquiryAllowed(p2.isLocationInquiryAllowed());
    } catch (boolean v0_18) {
        throw v0_18;
    }
    if (!this.isLocationInquiryAllowed()) {
        this.setLocationDeniedByUser(1);
    } else {
        if (!p2.isPublisherProvidingLocation()) {
            this.verifyLocationPermission();
            if (!this.isLocationDeniedByUser()) {
                this.switchOnLocUpdate();
            }
        } else {
            this.fillLocationInfo(p2.currentLocation());
        }
    }
    return;
}
```

Figure 17: Agreement analysis: code snippet 2