ARE YOUR SENSITIVE INPUTS SECURE IN ANDROID APPLICATIONS?

Trishla Shah, Raghav Sampangi and Angela Siegel

Dalhousie University, Halifax NS B3H 4R2, Canada

ABSTRACT

Android applications may request for users' sensitive information through the GUI. Developer guidelines for designing applications mandate that information must be masked/encrypted before storing or leaving the system. However not all applications adhere to the guidelines. As a prerequisite to tracking sensitive input data, it is essential to identify the widgets that request it. Previous research has focused on identifying the sensitive input widgets, but the extraction of all layouts, including images and unused layouts, is fundamental. In this paper, we propose an automated framework that finds sensitive user input widgets from Android application layouts and validates the masking of these inputs. Our design includes novel techniques for resolving the user semantics, extraction of resources, identification of potential data leaks and helping users to prioritize the sharing of sensitive information, resulting in significant improvement over prior work. We also train track the obtained sensitive input widgets and check for unencrypted transmission or storage of sensitive data. Based on a preliminary evaluation of our framework with some applications from the Google Play store, we observe notable improvement over prior work in this domain.

KEYWORDS

Android applications, sensitive, secure, GUI, layouts, framework.

1. INTRODUCTION

Applications may seek various user information through the graphical user interface (GUI), including sensitive user information such as credit card numbers, health card numbers, and social insurance numbers (SIN). Such applications must take precautions in data management and storage to protect sensitive information of users. To ensure that privacy and security requirements of users are appropriately met, it is important to vet such applications to identify abnormal behaviour.

Layouts and widgets are used to create the GUI in the Android application [1]. Layouts are containers that control widgets which are referred to as ViewGroup. Widgets are different UI components such as text boxes, labels, and buttons which are referred to as Views. These ViewGroups and Views are defined in XML files, and they may contain different attributes such as id, layout_height, and layout_width to present information to users and to collect user information. Attribute id is defined as "@+id/name", and it is unique for each widget in a layout. Developers may give any value to the IDs of such widgets [2], and, while against developer best practice guidelines, there is no current way to enforce such guidelines. This means that the IDs of widgets can take on any value and are not required to be related to the type of information being collected by the widget. Furthermore, the widgets may be placed in any order in the XML, which may be different from the order in which widgets are placed in the GUI layout, as shown in Fig.

David C. Wyld et al. (Eds): ICAIT, CBIoT, WiMo, CRYPIS, ICDIPV, CAIML, NLCA - 2022
pp. 73-89, 2022. CS & IT - CSCP 2022 DOI: 10.5121/csit.2022.121206

1. For example, the user sees the credit card text box below the "Enter credit card number" label, but it need not be written in the same order in the XML file as shown in Fig. 1. Due to the discrepancy in the name and order in which the labels and their respective input widgets are placed in the layouts, it becomes difficult to identify which information is asked in the respective input widget. Hence, analyzing the GUI of the applications and finding the correct widget that stores sensitive information opens up research directions in this area.

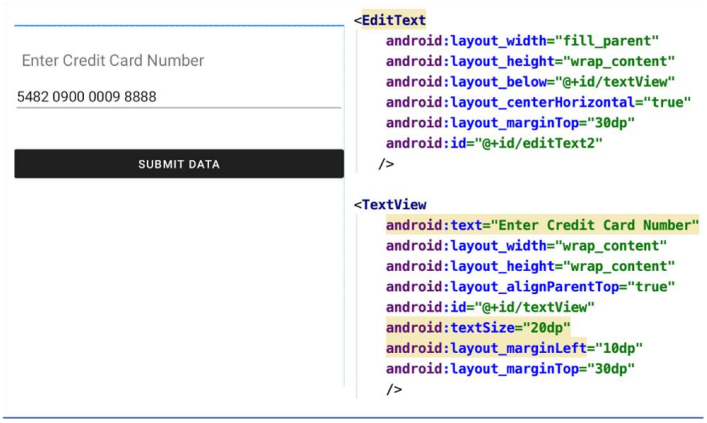

Figure 1. Sample layout and XML file

Prior research in this field has focused on resolving the widgets that seek sensitive user information [3-7]. However, such a focus on only sensitive user inputs may impact the ability to process the context and relevance of the underlying real-world information being collected, thereby affecting the semantic understanding of the context in which this information is being collected, and the overall accuracy of such observation. This motivated us to consider a deeper analysis of layout widgets, the identifiers used, the overall context in which this information is being collected, and to explore the possibility of weighting user inputs that are resolved using such a deeper analysis. This allowed us to conceptualize a framework that aims to provide users with insight about how their sensitive information is managed by an application, enabling them to make an informed decision about whether or not to share such information with the application.

Our proposed work enhances the accuracy of finding the sensitive user input widgets and advances the current state of the art by extracting more elements from the application's layouts and reducing the occurrence of false positives. We further train track these obtained input widgets to identify any sensitive data leaks. Our proposed framework will help users to make an informed decision when they use applications downloaded from the Google Play store or third-party Android app stores. While third-party Android app stores are not the recommended source to download apps [19], considering changes such as Google's announcement to increase the commission percentage for app developers [18] may consider switching venues in which their apps are made available. Such considerations pose a risk, which necessitates evaluation and mitigation of these risks prior to use of apps made available through third-party stores or the official Google Play Store.

In summary, the paper makes the following contributions:

- We develop a novel technique for analyzing the sensitive inputs of the user through the GUI improving the current state of the art (12% - 18% more accurate in detecting layouts over prior work and resolving all unused layouts in the application). We perform a direct comparison over the work done by SUPOR and UiRef by implementing their work.

- We obtain the sensitive information by analyzing the GUI of the applications and further train track this information to identify the security violations while the sensitive user data is within the application. Prior work has focused on using Data Dependency Graph (DDG) to identify security violation limited to payment data only. We design tests that apply for all the sensitive data of the users not just payment data.
- Our study highlights the loopholes in the design of current risk assessment tools (an experiment with Google's Play protect is conducted) and the need to take into account the user inputs in the security and privacy analysis of mobile applications.

The remainder of this paper describes the design and implementation of our technique to analyze the semantics of user inputs through GUI and train track of this information to identify potential security violations.

1.1. Motivation and Research Questions

Large Android application stores such as Google Play Store have implemented a "Play Protect" security feature to identify malicious applications [11]. Google Play Protect runs a safety check on applications from the Google Play Store before users download them. It also runs a safety check in the mobile phone on applications installed from outside of the Google Play Store. While running Play Protect on different sample applications which accepted and stored sensitive information such as credit card numbers, health card numbers, and SIN without masking, we found that Play Protect did not show an alert for these applications even if they were not masking user-sensitive information.

As shown in Fig. 2, we created a sample application that asked for sensitive user details such as credit card number, SIN, and passport number. The application was developed for testing purposes only and was not made available on any play stores for users to download. The sensitive user information captured by our sample application is not stored in encrypted form. It is a violation of the user's privacy and should have been detected [20]. When we passed the sample application to Google's Play Protect, it did not raise any security alerts, as shown in Fig. 2. The experiment helped us to understand the current state of art of risk assessment systems and the need to protect users' sensitive information.

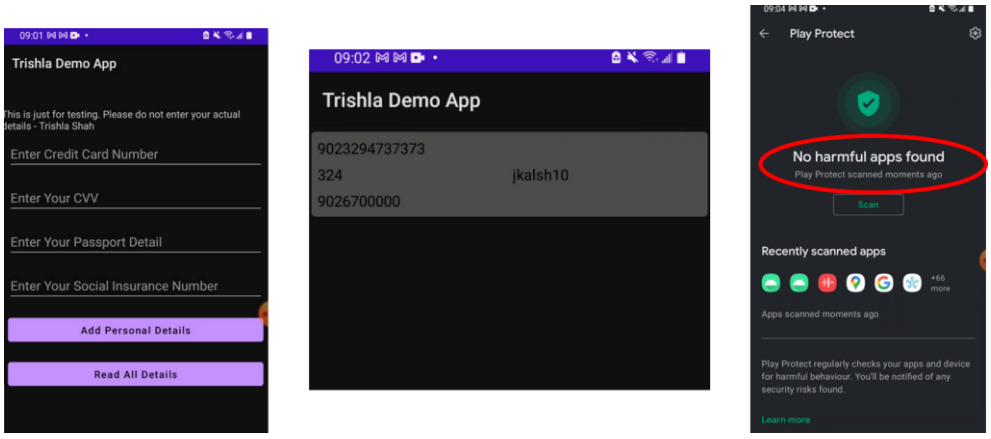

Figure 2. Google Play Protect Experiment

This prompted us to consider developing a framework that would analyze applications and identify sensitive user information, its masking, and storage of sensitive authentication data.

1.1.1. Research Questions

Based on the simple experiment conducted in the previous section, we understood the impact of poorly defined semantics in designing the application's layout. Hence, it is crucial to protect the user's sensitive data and verify that it follows appropriate privacy and security standards.

Our work focuses on automatically resolving the sensitive user inputs in Android applications through the GUI, and further train track this information to identify the potential security risks associated with it. We also calculate the weights of the identified inputs which helps to prioritize the sharing of sensitive information and inform decision making.

The following research questions guided the work:

1. Which input fields seek sensitive information?
2. Is the sensitive information masked before storing or leaving the system?
3. Is the sensitive authentication data stored after authorization?
4. For each detected input widget, what is the level of sensitivity measured by the system?

To answer the above research questions, we started our analysis by conducting a detailed literature survey of existing work in this field.

2. RELATED WORK

We previously researched the occurrence of a relay attack during credit card payments at the point of sale (POS) and designed protocol to prevent relay attack [17]. Our research then extended in the direction of securing sensitive user input data shared through the GUI of Android applications. Our work is not the first in this area of resolving user inputs and further train track the obtained information. Prior research has attempted to identify user input widgets through analysis of the GUI of Android applications [3-7] as shown in Table. 1.

Recent research in this direction [7] focuses on understanding the intentions of the icons and the sensitive GUI widgets in Android applications. However, they do not consider the problem of word ambiguity which leads to a drastic drop in the accuracy of detecting the sensitive widgets. Also, the set generated for sensitive terms and categories of user data is very limited. FlowCog [8] focuses on context-aware semantic extraction and analysis of information flow leaks in Android applications. When extracting the layouts for such semantic analysis, unused layouts must be considered to reduce the occurrence of false positives. However, no technique is incorporated in [8] to address the issues with unused layouts as well as word ambiguity.

UiRef [3] has shown better results in addressing user input semantics; however, it faces the issue of occurrences of false positives in the final result due to unused layouts in the candidate set. Also, UiRef does not consider images while extracting layout elements resulting in incomplete extraction of resources which leads to inaccurate results [3]. SUPOR[4], UIpicker [5], and AppsPlayground [6] also attempted to detect precise and scalable sensitive user input, but did not address the issue of word ambiguity and the occurrence of false positives due to unused layouts. Moreover, they do not handle custom layouts of applications which leads to insufficient analysis as most of the current applications use custom views for designing their GUI. All the existing research focuses on the extraction of resources for static layouts and do not consider dynamic layouts in their analysis. Also, no existing research in this domain focuses on measuring the weights of the detected sensitive input by the framework.

Table 1. Identification of user inputs requesting sensitive information.

Research	Primary Goal	Open Research Gaps
IconIntent (2019) [7]	Understanding the intentions of icons and identifying sensitive GUI widgets	1) Only detects sensitive icons, 2) Relies on SUPOR to identify sensitive text widgets, 3) does not consider word ambiguity problem
FlowCog (2018) [8]	Context-aware semantics extraction and analysis of information flow leaks	Does not address problem of word ambiguity and unused layouts
UiRef (2017) [3]	Resolving the sensitive user inputs	Does not considers unused layout and extraction of images
SUPOR (2015) [4]	Precise and scalable sensitive user input detection	Does not consider unused layout and extraction of images. Also, it does not address problem of word ambiguity and custom layouts
UIPicker (2015) [5]	User-input privacy identification	Same as SUPOR
AppsPlayground (2013) [6]	Automatic security analysis of smartphone applications	Same as SUPOR

To answer our research questions RQ2 & RQ3, we study the existing literature that focus on identifying and verifying the privacy and security constraints of sensitive user information in Android applications. This is achieved by tracking the flow of sensitive information in Android applications and identifying potential data leaks and security violations. These violations include 1) unencrypted or unmasked storage of sensitive user input within the device, 2) unencrypted or unmasked transmission of sensitive user input from the application, and 3) storage of authentication data within the device such as CVV.

Prior work in this direction (as shown in Table. 2) has focused only on payment data, and the test cases derived are specific to Payment Card Industry Data Security Standard (PCI DSS) [16]. Cardpliance showed promising results in detecting security violations for payment data [14]. However, it showed dependency in obtaining the user inputs on work done by UIRef. This leads to a few inaccuracies, as mentioned by UiRef, one of which is undetected user inputs. Table. 2 outlines the open research gaps by the existing work in this direction and we address it in the following subsection.

Table 2. Identification and notification of any potential data leak

Research	Primary Goal	Open Research Gaps
Cardpliance (2020) [14]	PCI DSS Compliance of Android applications	It does a keyword-based search to identify inputs related to the credit card from the list obtained from the UiRef module. It only focuses on credit card-related inputs. It relies on UiRef, which implies that it does not consider image. Also, it does not address if any extra identifiers may exist because of unused layouts which were not removed in UiRef.
FlowDroid(2014) [15]	Static taint analysis for Android applications.	Difficult to extend and configure custom test.

2.1. Research Gaps

We have identified the following research gaps based on our literature survey:

1. Prior techniques do not consider images and dynamic layouts when resolving user input widgets. These elements must be considered to improving the accuracy of detection.
2. Unused layouts must be removed to reduce the occurrence of false positives.
3. The problem of multiple-input widgets for a single label needs to be resolved to address the issue of incorrect pairing of label and text input.
4. The framework must assign some weightage to the final result which helps users to prioritize the sharing of sensitive information and to make an informed decision.
5. The above research gaps opened up directions where possible contributions can be made in this field.

3. PROPOSED WORK

Our work focuses on automatically resolving the sensitive user inputs in Android applications through GUI and further train track this information to identify the potential data leaks, achieving three main goals. First, we identify any types of sensitive information that the application requests. Sensitive information is any input that reveals users' personal information such as health records, payment information, SIN, and passport number. Second, we identify if the application follows appropriate Android developers' security and privacy guidelines [20] while storing or transmitting the requested sensitive information to avoid data leaks. Third, we assign weights to the obtained results, representing our detection strength. It will help users prioritize sharing sensitive information with a particular application. It also enables users to make informed decisions. Our framework is as shown in Fig. 3.

We divide our work into three sections: 1) Identification of user inputs requesting sensitive information 2) Identification and notification of any potential data leaks, and 3) Assigning weights to the obtained sensitive inputs.

3.1. Identification of user inputs requesting sensitive information

We focus on resolving the data semantics of user input widgets in Android applications by analyzing the GUI of these applications. As shown in Fig. 3 the module 1 has three main components: Extraction and optimization of resources, filtering of sensitive labels and mapping of sensitive labels with relevant user input. Extraction and optimization of resources will extract the layouts and exports the rendered layouts and resources for further analysis. We then filter the sensitive labels from the extracted layouts and resources, which is analyzed further for mapping of sensitive labels with the input widgets. We also apply text analytics technique to extract the sensitive labels and its context to a given layout.

For the identification of user inputs that request sensitive information (Module-1), we bifurcate our work into the following segments: 1) Extraction and optimization of resources, 2) Filtering of sensitive labels, and 3) Mapping of sensitive labels with relevant user inputs

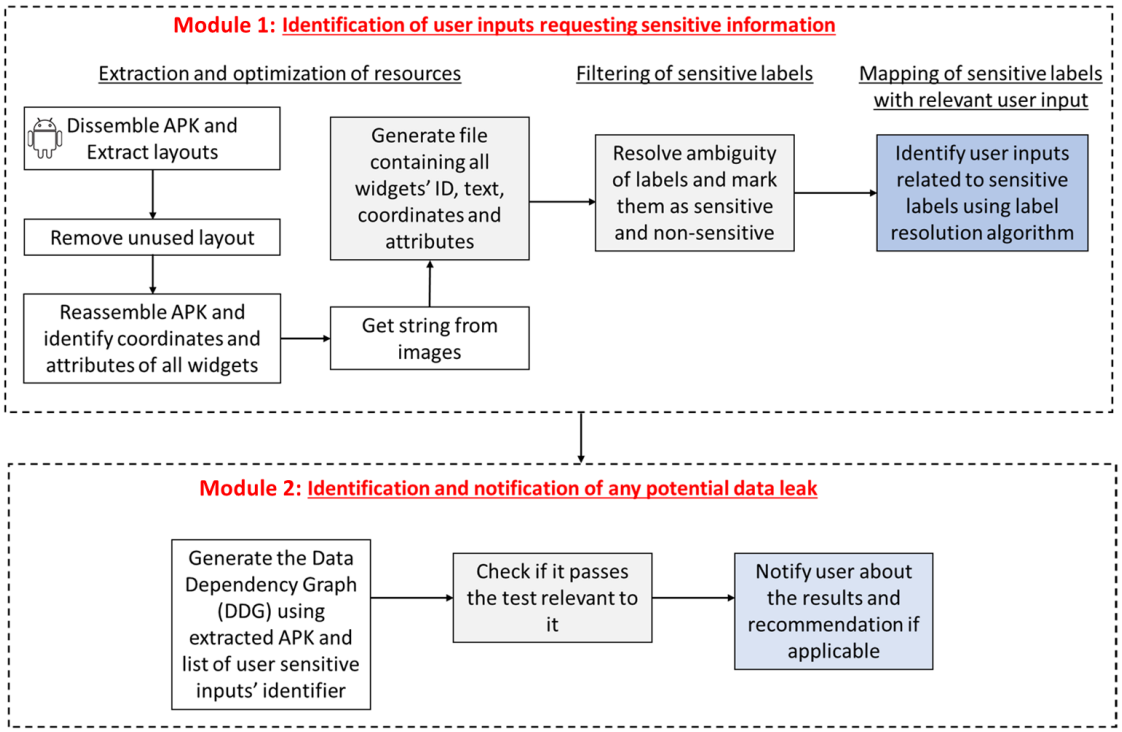

Figure 3. Proposed framework

3.1.1. Extraction and optimization of resources

In Android, there are different types of layouts: linear layout, constraint layout (relative layout), and frame layout. Each of these layout will have a different placement of the labels and input widgets (as shown in Fig. 4) The developer uses these layouts to design the GUI of the application. Whenever a layout is created, its corresponding XML file also gets generated (as shown in Fig. 4).

Statically parsing the XML file to extract the widgets is one of the methods. However, only parsing the XML file statically can lead to undetected widgets as most applications use a custom view [3]. Android does not force the developers to place widgets in a specific order and only provides guidelines. Hence, it is not easy to find the position of widgets based on the order in which the user sees them. In our work, we examine the XML files statically to retrieve all widgets and then render these widgets on the Android virtual device (AVD) to identify their coordinates which will be used further to map labels and sensitive user inputs.

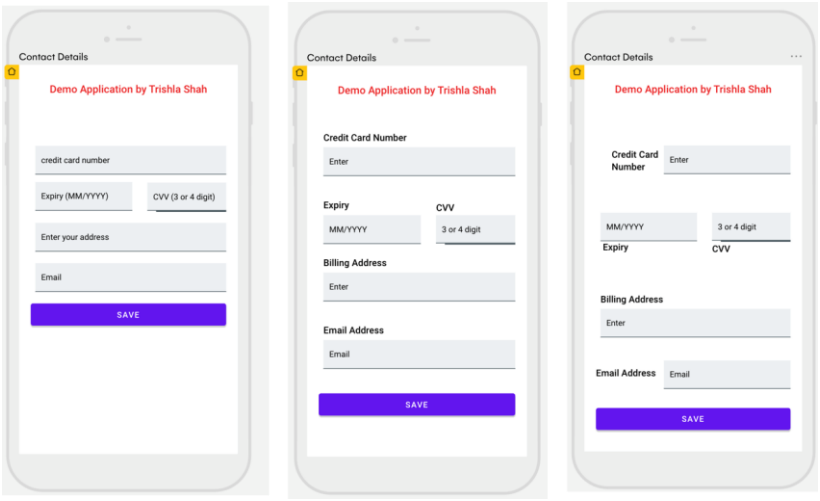

Figure 4. Different types of layouts

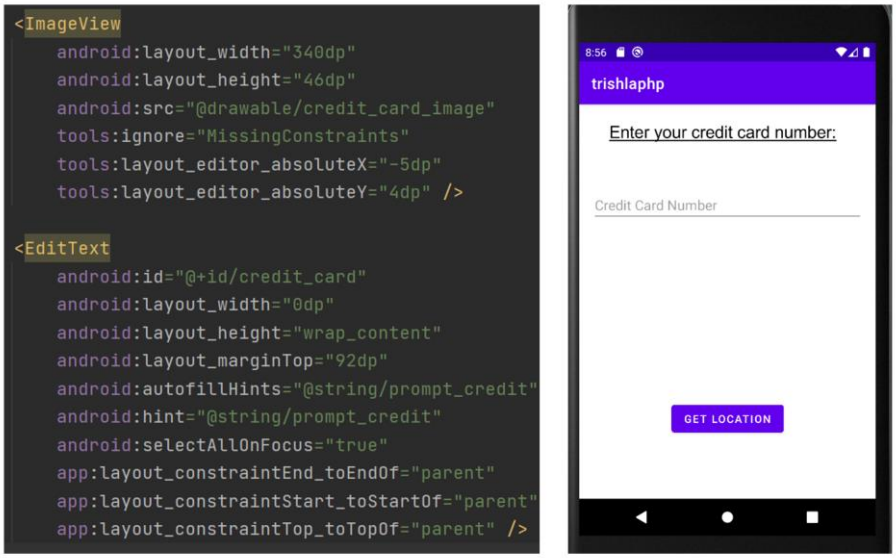

Figure 5. EditText with a label as an ImageView

The goal of this module is five-fold:

1. Extracting all the layouts from the applications
2. Removing unused layouts
3. Identifying widgets such as EditText and labels
4. Identifying images and extracting the text from them
5. Retrieving coordinates of all widgets

While developing the application, there may be a few layouts that the developers design but are not used when the application is executed. These layouts are known as unused layouts. During our initial analysis, we found that most of the current applications have unused layouts in the application package. Prior research [3] shows that the presence of unused layouts increases the occurrence of false positives, which reduces the overall accuracy of sensitive user input detection.

Our layout extraction module focuses on removing the unused layouts, thus decreasing the occurrence of false positives.

Another challenge while extracting resources is identifying images that are used as labels for prompting user to enter sensitive information. For example, the widget used in Fig. 5 looks like a text view – “Credit Card” but when the XML file is analyzed, the text displayed is actually an image. These images would be discarded from the analysis if we only focus on extracting widgets from the layouts. We convert the images to text and then extract it as a part of our analysis, which no prior research has focused on.

The extraction process is done by disassembling the APK using APKTool and extract all layout files and resources [9]. Developers may compile and publish applications without cleaning them. After disassembling, we first remove any unused layouts using Android Lint tool [12]. Android Lint is a static analysis tool that optimizes the application by removing unused layouts. If the layout file or widget is present and not referenced anywhere, we consider it as an unused layout file or widget. For example, two credit card input widgets are present with different styles, but only one credit card input widget is used in the application. We then remove the widget/layout which is not used by the application during execution. This process helps us to address the issue of unused layouts and remove them from our analysis.

The next process is to extract the images used in the layouts and convert them into strings using optical character recognition (OCR) techniques. This helps to reduce the false negatives. We use pytesseract OCR tool to read the text embedded in images [13]. We take into consideration formats of images such as jpeg, png, gif, bmp, and tiff. We then create list of widgets used in the application and their resource identifiers. Once our resource identifiers are collected, we add one custom activity file, which renders all collected resource identifiers using setContentView(). We use APKTool to recompile the APK file with payload (list of resource identifiers) and custom activity. We then replace the old signature with the new one and align the new APK using APKSigner [10]. Then we run the newly generated APK on the AVD, which will extract coordinates, visibility attributes, hint, and text strings of all widgets. This information will be stored in a separate file with their identifier and used in the next module.

3.1.2. Filtering of sensitive labels

This module aims to resolve the semantics of the input data type and the descriptive text associated with it. We only focus on the sensitive terms and the associated descriptive text. It is crucial to identify the correct terminology and concept of the associated input data type. These can be single words or phrases. While mapping different terms to the same concept, there are two issues identified by previous literature [3]. These issues are: Synonymy - different terms may represent the same concept and Polysemy - same terms may represent different concept [21]. While extracting sensitive terms from the layouts, the concept of polysemy and synonymy helps to understand the context of the associated text. It is important to extract single words, phrases and resolve the concept that each of these sensitive terms relate to. This helps to reduce the ambiguity of words. We filter each of these sensitive terms (single words and phrases) by extracting them individually and comparing it with our dictionary of sensitive terms.

The process starts by lemmatizing all extracted text of labels, identifiers, hints and removal of stopwords. We then replace the special characters with the associated words. For example, if we find #, we replace it with a "number" word. The next step is to identify the sensitive terms by doing a word match with the predefined list and extracting it. This predefined list is developed by collecting inferences from prior work [4] and the Android guidelines for sensitive information [20]. If a match exists within the list, the label will be marked as "sensitive", "non-sensitive"

otherwise. There is a possibility that some of the labels are sensitive but could not be matched because of ambiguity. For example, address can be used for "postal address" and for" IP address". Hence, when address is matched, it is important to resolve the concept to which it refers and then labelling it as "sensitive" and "non-sensitive". To protect sensitive information, we must remove ambiguity from these words. Before we disambiguate the results it is important to understand the concept i.e. the different meanings of a particular term (polysemy or synonymy).

We use the word sense disambiguation method to resolve ambiguous words. Word sense disambiguation helps us to resolve meaning of a specific term. While performing the word match of individual terms/phrases, we perform the word sense disambiguation method to resolve the concept of the identified terms. We collect all the text within the layout to find the semantic group of the matched term/phrase. We also maintain a list of possible ambiguous words by exploring different Android applications, and those collected by prior work [3]. This helps to perform a word sense induction on the terms that are ambiguous but not identified as "sensitive". The concept of the ambiguous words is resolved as shown in Fig. 6.

Once we have the set of terms that relate to the ambiguous words on hand, we calculate the score for those participating terms . For example, the words such as city, postal code, and country will cause it to replace the address with "postal address" as it will have the highest score. We use the word only once for the calculation even if it appears multiple times. If two groups have the same score, then we calculate the score based on the number of times each word appears. If "city" appears three times, then we assign a score of 3 to it. If we get the same score for the two participating groups, then we keep both the groups. For example, if the postal address and IP address have the same score, it will show "postal address or IP address" after the resolution. We will mark it sensitive if at least one of the groups is from our sensitive word list.

Since we rely on the text collected from the layout, it does not disambiguate the word if the layout has only one label in which the ambiguous word is present. It is fair to assume to make that layout with a single widget will not have any ambiguous words present, since it will make it difficult for users to enter the required information in the input widget. This limitation can be addressed in future work by considering all layout files to resolve the ambiguous words.

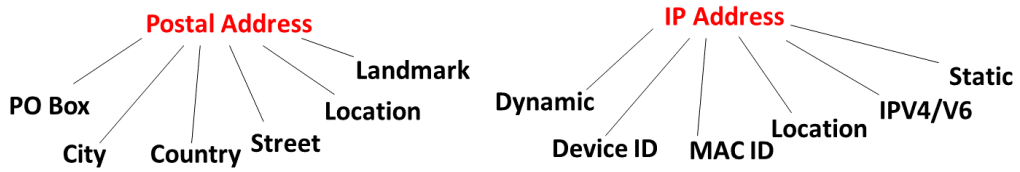

Figure 6. Sample ambiguous word groups

At the end, we remove all non-sensitive labels. However, we consider all the input widgets from the layouts without discarding any as it helps in the later stage to match the sensitive labels with all the extracted input widgets. Removing all non-sensitive labels helps to avoid incorrect pair-value matching, thus improving accuracy.

3.1.3. Mapping of sensitive labels with relevant user inputs

The main task of this module is to identify the input widgets to which the sensitive labels refer. Identifying the input widgets helps to retrieve the sensitive value entered by the user and thus train track it in the later stages of our framework. The first step is to identify the input widgets

that are related to the matched sensitive labels. To achieve this, we pass the generated sensitive labels from the semantic resolution module and find its corresponding widgets.

We devise two different algorithms to identify pairs of input widgets and sensitive labels. The two algorithms are as follows:

1. Generate candidate sets and vectors
2. Find the optimal set

Algorithm 1 - Generate candidate sets and vectors: The algorithm starts by generating candidiate sets of all the sensitive labels and input widgets. It then creates a set of vectors which represent the Euclidean distance within the GUI and direction from each sensitive label to all potential input widgets in the layout. The method calcSmallestVector() of UiRef [3] is used to generate the three smallest vectors that represent the potential distance of the sensitive term with the input widget. In this method, two vectors are generated from the two closest corners of the label to corresponding corners of the input widget. If the label is directly above, below, or on the sides of the input widget, then a third vector is created. A vector will not be considered as a candidate if the Euclidean distance is greater than a defined threshold, which is based on the screen size of the device being used to run the application. We obtain a set of potential candidates based on the Euclidean distance as the end result for this algorithm.

Algorithm 1 Algorithm for generating candidate sets and vectors

Input: All labels ($label$) and input widgets (iw)
Output: Candidate sets (cs)
$Initialisation : Candidates, v$
1: **if** ($i \neq$ "$sensitive$") **then**
2: Discard i
3: **else**
4: **for each** $i \in label$ **do**
5: **for each** $l \in iw$ **do**
6: $v \leftarrow calcSmallestVector(i, l)$
7: **if** ($v.distance < threshold$) **then**
8: $candidates[v].append(i, l)$
9: **end if**
10: **end for**
11: **end for**
12: **end if**
13: **return** $Candidates$

Figure 7. Algorithm 1 - Generate candidate sets and vectors

Algorithm 2 - Find the optimal set: The list of candidate sets obtained from Algorithm 1 will be passed as input to Algorithm 2, which will attempt to find the optimal set. It starts from the largest candidate set. It selects a label to input-widget pair if the input widget is directly above, below or on the side of the label. If there are multiple input widgets directly above, below or on the side of the labels, then it will choose the input widgets with the smallest distance. This process is repeated until it resolves all sensitive labels and finds the input widget pairs for it.

Algorithm 2 Algorithm to find the optimal set

Input: All labels ($label$), candidate sets (cs)
Output: Resolved labels ($ResolvedLabels$)
$Initialisation : ResolvedLabels, CountSensitive = 0$
1: **for** $i = 0$ to $size(labels)$ **do**
2: **if** $(i == "sensitive")$ **then**
3: $CountSensitive = CountSensitive + 1$
4: **end if**
5: **end for**
6: **while** $CountSensitive > 0$ **do**
7: $pairs \leftarrow GenCandidateSets(label, iw)$
8: $labels.remove(pairs.labels)$
9: $iw.remove(pairs.iw)$
10: $ResolvedLabels.append(pairs)$
11: $CountSensitive = CountSensitive - 1$
12: **end while**
13: **return** $Candidates$

Figure 8. Algorithm 2 – Find the optimal set

Towards the end of this module, we will have the sensitive inputs and its corresponding widgets which helps us to retrieve the value of sensitive input given by the user. Once we have received the sensitive labels and its corresponding widget pairs, we analyze this sensitive user information and train track it to identify security violations and possible data leak.

3.2. Identification and notification of any potential data leaks

This module is guided by the research questions " Is the sensitive information masked before storing or leaving the system?" and "Is the sensitive authentication data stored after authorization?". To answer the above research questions, we need to understand the technical approach to solving the problem and challenges associated with it. First, which requirements are considered for protecting the sensitive data in mobile applications? Second, how can these requirements be translated into program analysis tasks while minimizing false positives? Third, how can the violations associated with encryption/masking and authorization of sensitive data be programmatically identified?

In this module, we design our own static analysis method that captures the requirement of protecting user sensitive data in mobile applications and notify for potential security violations within the applications. Prior work in this domain [14] has focused on designing the static analysis tool but is limited to only payment applications. We consider identifying the potential data leaks within all the Android applications that seek for sensitive user information.

We identified the security requirements for sensitive user data in Android applications which are as follows:

- R1: Sharing of data securely across applications.
- R2: Storing of data safely within the application.
- R3: Limit the sensitive information storage and retention time.
- R4: Store only non-sensitive data in cache files.

- R5: Enforce secure communication.

Module-2 in (Fig 3) shows the high level overview of our proposed methodology in identifying the security violations of sensitive user data in mobile applications. We obtain the sensitive labels and widget pairs from Module-1 of the proposed framework and further train track this information. The next phase includes building a Data Dependency Graph (DDG) to identify the flow and context sensitive static program analysis on the .apk files. We use insights from Cardpliance [14] and choose Amandroid [22] as our application analysis tool. Amandroid allows us to perform static analysis and produce graphs upon which these tests can be performed. The sensitive input widgets received from Module-1 of the proposed framework will be passed to Amandroid that handles this information based on the tests designed and tracks the data flow of the GUI input.

We designed our own tests by taking reference from Cardpliance [14] for identifying violations and possible data leaks. These tests are built on top of Amandroid [22] which helps to perform the static analysis and generate the alerts for violations. The tests are as shown in Table. 3.

Table 3. Tests to identify violations and possible data leaks

Tests	**Identifies**	**Source (S)**	**Sink (K)**	**Required Methods (R)**
T1	Check if it is storing data or not	Activity.findViewById(ID)	DPM	-
T2	Check whether it is masking	Activity.findViewById(ID), URLConnection.getInputStream()	View.setText()	MM
T3	Storing non-obfuscated data	Activity.findViewById(ID)	DPM	OM
T4	Sharing Non-Obfuscated data	Activity.findViewById(ID)	Intent.putExtra(), SmsManager.sendTextMessage()	OM

Data Persistence Methods (DPM): java.io.OutputStream.write(), java.io.FileOutputSream. write(), java.io.Writer.write(), java.lang.System.out.println(), android.content.SharedPreferences Editor.putString(), android.util.Log.i(), android.util.Log.d()

Masking Methods (MM): java.lang.String.replace(), java.lang.String.substring(), java.lang. String.concat(), java.lang.StringBuilder.append()

Obfuscation Methods (OM): javax.crypto.Cipher.update(), javax.crypto.Cipher.updateAAD(), javax.crypto.Cipher.doFinal(), java.security.MessageDigest.digest(), java.security.MessageDigest .update()

These tests will be passed to our framework and tested on our dataset of Android applications. For the scope of this paper, we do not evaluate the results for this module of our framework.

3.3. Assigning weights to the obtained candidate sets

The goal of our framework is to inform users of the potential violations and data leaks of their sensitive information. However, it is also essential to inform the user of how confident we are about the result. Our framework might give false-positive or false-negative results in some scenarios, such as multiple ambiguous words in a single layout. For example, our framework identified that the application is asking for a credit card number because it found the "number"

keyword and a few other related terms. Hence, it will give a "very low" weight because the framework did not find any direct evidence that the input is for credit card information. If we find "credit card" or a similar term in the hint attribute of the input widget and the related label, then it will give a "very strong." weight as shown in Table. 4.

For the scope of this paper, we share the initial idea for assigning weights and do not discuss its methodology and evaluation results.

Table 4. Assigning weights to the obtained candidate sets

Scenario	**Weight**
If it finds sensitive user input using hint and label or ID	Very strong
If it finds sensitive user input using hint only	Strong
If it finds sensitive user input using both label and ID	Low
If it finds sensitive user input only label or only ID	Very low

4. EVALUATION

In this section, we evaluate the effectiveness of our framework with respect to the Module-1 of our framework (i.e., Identification of user inputs requesting sensitive information.)

To evaluate our extraction and optimization of resources module, we use some emulators and real time devices. We use 768 x 1280: xhdpi Nexus 4 emulator, running Android 10.0.

Our dataset consists of 60 Android applications from different categories of Google Play as shown in Table 5. We have created 10 of our own sample applications out of 60 applications and marked as "Other" to test different scenarios. To verify these applications' language, we use Python's langdectect module [3] for ensuring that applications with English descriptions are chosen.

Table 5. Applications for the evaluation

Application Category	**# of Apps**	**# of input**	**# of labels**	**# of images**	**# of hints**
Finance	10	128	203	41	88
Health & Fitness	10	103	143	34	76
Education	10	48	51	25	43
Entertainment & Travel	10	79	102	62	72
Shopping	10	112	214	84	79
Other	10	118	183	66	48
Total	**60**	**588**	**896**	**312**	**406**

We use UiRef and SUPOR [3, 4] as our baseline for comparison. We had no access to SUPOR's source or binary code and hence we re-implement their approach from the information gained by their publications. With UiRef, we were able to retrieve the loose code and modify it based on our comparison parameters.

Table 6. Extraction & optimization of resources

Parameters	**SUPOR**	**UiRef**	**Our Approach**
No. of widgets detected (Out of 2202)	1658	1790	2047
% of detection	75.29%	81.29%	92.96%
Unused widgets detected	N/A	N/A	103
Images to text conversion (Out of 312)	N/A	N/A	198 (63.46%)

Table 7. Filtering of sensitive labels

Parameters	**SUPOR**	**UiRef**	**Our Approach**
No. of ambiguous words detected (Out of 10)	N/A	9	9
% of ambiguous words detected	N/A	90%	90%
No. of labels (we also consider images as labels) detected (Out of 1208)	848 (70.19%)	876 (72.51%)	1074 (88.90%)
No. of sensitive labels detected	N/A	N/A	204

Table 8. Mapping of sensitive labels with relevant user inputs

Parameters	**SUPOR**	**UiRef**	**Our Approach**
No. of pairs detected - Only sensitive (Out of 206)	148	196	197
% of detection	78.72%	95.14%	96.11%
False positive (No. of sensitive pairs)	13	11	3
Accuracy in %	65.53%	89.80%	94.17%

The parameters based on which the comparison is done are chosen with the following aspects:

1. Number of elements detected has a direct correlation with the accuracy of detecting the correct input widgets. As greater no. of elements are detected, the initial dataset becomes more detailed. Our end goal is to detect correct pairs of the widgets wherein the user might enter the sensitive input. For this detection, all, or most of the elements of the layout must be detected. If the no. of elements detected is less than the actual elements present in the application, then the widget pairs generated will also be limited thus reducing accuracy of detection. As can be seen in Table. 6, our framework is detecting 12-18% more widgets than the prior work.
2. Detecting unused layouts is crucial as its presence has a direct impact on the ratio of false positive. Unused layout detection is one of our contributions and is not attempted by prior work. Hence the result comparison for this parameter is not applicable to the prior work. Unused widgets are around 5% of total widgets (as shown in Table. 6.
3. Converting images to text is one of our contributions and not attempted by prior work. Hence the result comparison for this parameter is not applicable to the prior work. We were able to convert 63.46% of the images to text. Some images did not have any text, so we could not be able to convert those. We found that accuracy can be improved by including icons, such as VISA or MasterCard, to detect the credit card field. We plan to include icons in our future work.
4. For the filtering of sensitive labels module, our main goal is to filter sensitive labels. For filtering sensitive labels, we first build a dictionary of possible sensitive terms and then do a text-matching. The same process is also carried out by prior work. However, the accuracy of this detection is based upon the ability of the algorithm to resolve ambiguous

words. Hence, the number of ambiguous words detected is crucial in determining the accuracy for this module. We had very few ambiguous words in the chosen applications. We were able to detect and resolve 90% of ambiguous words. UiRef also detected 90% of ambiguous words. Both (UiRef and our method) failed to resolve ambiguous words when only one layout and very few labels were available.

5. Prior work has focused on detecting all the labels – sensitive and non-sensitive. We first detect all labels and remove any non-sensitive labels from the list. We have included images as labels since some images contain text in them. As shown in Table. 7, we dected 18.5% - 21% more labels as compared to prior techniques.
6. Mapping sensitive labels with relevant user inputs modules generate the final candidate sets wherein the user inputs for sensitive labels are stored. The number of pairs detected does not have a correlation with the accuracy. However, the number of pairs detected will give an insight into the initial dataset generated and how many candidate sets the algorithm was able to generate. UiRef and our method detected almost the same number of pairs, as shown in Table. 8, but UiRef does not remove any unused layouts or widgets. So it is very important to identify the number of false positives and false negatives when determining the algorithm's overall accuracy. We found a 4.6% - 30% improvement in the accuracy from the prior techniques.

5. CONCLUSION AND FUTURE WORK

Our studies focus on resolving the semantics of sensitive user input through the GUI in Android applications and the train track of this information to identify potential violations and data leak. Prior studies have focused only on solving a part of this problem (either sensitive inputs or train track of information) and largely ignored the violations associated with sensitive information as a whole. In this paper, we have presented methodologies for identifying the input widgets that seek sensitive user information in Android applications and further train track of this information to identify potential data leaks and security violations. We also assign weights to the obtained results from our Module-1 which helps the users to prioritize the sharing of sensitive information through the application. This helps users to make an informed decision. No prior research in this field has attempted to provide weights to the obtained results. With our proposed framework, we are able to achieve an improvement in accuracy of 4.6 % - 30%. Our findings demonstrate that understanding the security and privacy concerns of user's sensitive inputs provide a unique vision to the mobile application's data security.

REFERENCES

[1] “Build a simple user interface,” Android, [Online]. Available: https://developer.android.com/training/basics/firstapp/building-ui. [Accessed 13 May 2022].

[2] “Layout resource,” Android Developers, [Online]. Available: https://developer.android.com/guide/topics/resources/layout-resource. [Accessed 13 May 2022].

[3] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh and T. 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, 2017.

[4] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang and G. Jiang, “SUPOR: Precise and scalable sensitive user input detection for android apps,” in 24th USENIX Security Symposium (USENIX Security 15), 2015.

[5] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu and X. Wang, “UIPicker: User-Input Privacy Identification in Mobile Applications,” in 24th USENIX Security Symposium (USENIX Security 15), 2015.

[6] V. Rastogi, Y. Chen and W. Enck, “Appsplayground: automatic security analysis of smartphone applications,” in Proceedings of the third ACM conference on Data and application security and privacy, 2013.

[7] X. Xiao, X. Wang, Z. Cao, H. Wang and P. Gao, "Iconintent: automatic identification of sensitive ui widgets based on icon classification for android app," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019.
[8] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao and Y. Chen, "FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps," 2018.
[9] Apktool, "Apktool - A tool for reverse engineering 3rd party, closed," [Online]. Available: https://ibotpeaches.github.io/Apktool/. [Accessed 14 May 2022].
[10] "Apksigner," Android Developers, [Online]. Available: https://developer.android.com/studio/command-line/apksigner. [Accessed 14 May 2022].
[11] "Play Protect," Google Developers, [Online]. Available: https://developers.google.com/android/play-protect. [Accessed 14 May 2022].
[12] "Improve your code with lint checks - Android Developers," Android Developers, [Online]. Available: https://developer.android.com/studio/write/lint. [Accessed 14 May 2022].
[13] M. Lee, "pytesseract: Python-tesseract is a python wrapper for Google's Tesseract-OCR," PyPI, [Online]. Available: https://pypi.org/project/pytesseract/. [Accessed 14 May 2022].
[14] S. Y. Mahmud, A. Acharya, B. Andow, W. Enck and B. Reaves, "Cardpliance:PCI DSS Compliance of Android Applications," in 29th USENIX Security Symposium (USENIX Security 20), 2020.
[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," Acm Sigplan Notices, vol. 49, pp. 259-269, 2014.
[16] "Official PCI Security Standards Council Site - Verify PCI Compliance, Download Data Security and Credit Card Security Standards.," [Online]. Available: https://www.pcisecuritystandards.org. [Accessed 14 May 2022].
[17] T. Shah and S. Sampalli, "Efficient LFSR Based Distance Bounding Protocol for Contactless EMV Payments," in Proceedings of the Future Technologies Conference, 2018.
[18] "Service fees - Play Console Help.," Android Developers, [Online]. Available: https://support.google.com/googleplay/android-developer/answer/112622?hl=en. [Accessed 14 May 2022].
[19] B. Uscilowski, "Mobile adware and malware analysis," in Symantec Corp, 2013.
[20] "User Data - Play Console Help.," [Online]. Available: https://support.google.com/googleplay/android-developer/answer/10144311. [Accessed 14 May 2022].
[21] D. Glynn and J. A. Robinson, "Corpus methods for semantics: Quantitative studies in polysemy and synonymy," 2014.
[22] F. Wei, S. Roy and X. Ou, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," ACM Transactions on Privacy and Security (TOPS), vol. 21, pp. 1-32, 108.