SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps

Jianjun Huang, *Purdue University;* Zhichun Li, Xusheng Xiao, and Zhenyu Wu, *NEC Labs America;* Kangjie Lu, *Georgia Institute of Technology;* Xiangyu Zhang, *Purdue University;* Guofei Jiang, *NEC Labs America*

This paper is included in the Proceedings of the 24th USENIX Security Symposium

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps

Jianjun Huang[1], Zhichun Li[2], Xusheng Xiao[2], Zhenyu Wu[2], Kangjie Lu[3], Xiangyu Zhang[1], and Guofei Jiang[2]

[1]Department of Computer Science, Purdue University
[2]NEC Labs America
[3]School of Computer Science, Georgia Institute of Technology
huang427@purdue.edu, {*zhichun, xsxiao, adamwu*}*@nec-labs.com, kjlu@gatech.edu*
xyzhang@cs.purdue.edu, *gfj@nec-labs.com*

Abstract

While smartphones and mobile apps have been an essential part of our lives, privacy is a serious concern. Previous mobile privacy related research efforts have largely focused on predefined known sources managed by smartphones. Sensitive user inputs through UI (User Interface), another information source that may contain a lot of sensitive information, have been mostly neglected.

In this paper, we examine the possibility of scalably detecting sensitive user inputs from mobile apps. In particular, we design and implement SUPOR, a novel static analysis tool that automatically examines the UIs to identify sensitive user inputs containing critical user data, such as user credentials, finance, and medical data. SUPOR enables existing privacy analysis approaches to be applied on sensitive user inputs as well. To demonstrate the usefulness of SUPOR, we build a system that detects privacy disclosures of sensitive user inputs by combining SUPOR with off-the-shelf static taint analysis We apply the system to 16,000 popular Android apps, and conduct a measurement study on the privacy disclosures. SUPOR achieves an average precision of 97.3% and an average recall of 97.3% for sensitive user input identification. SUPOR finds 355 apps with privacy disclosures and the false positive rate is 8.7%. We discover interesting cases related to national ID, username/password, credit card and health information.

1 Introduction

Smartphones have become the dominant kind of end-user devices with more units sold than traditional PCs. With the ever-increasing number of apps, smartphones are becoming capable of handling all kinds of needs from users, and gain more and more access to sensitive and private personal data. Despite the capabilities to meet users' needs, data privacy in smartphones becomes a major concern.
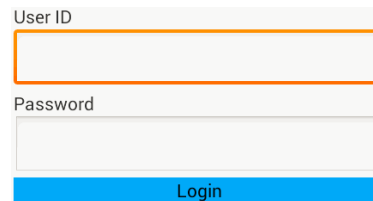

Figure 1: Example sensitive user inputs.

Previous research on smartphone privacy protection primarily focuses on sensitive data managed by the phone OS and framework APIs, such as device identifiers (phone number, IMEI, etc.), location, contact, calendar, browser state, most of which are permission protected. Although these data sources are very important, they do not cover all sensitive data related to users' privacy. A major type of sensitive data that has been largely neglected are the *sensitive user inputs*, which refers to the sensitive information entered by users via the User Interface (UI). Many apps today acquire sensitive credentials, financial, health, and medical information from users through the UI. Therefore, to protect and respect users' privacy, apps must handle sensitive user inputs in a secure manner that matches with users' trust and expectations.

Figure 1 shows an example interface an app uses to acquire users' login credentials via input fields rendered in the UI. When users click the button "Login", the app use the user ID and password to authenticate with a remote service. As the developers may be unaware of the potential risk on the disclosures of such sensitive information, the login credentials are sent in plain text over an insecure channel (HTTP), which inadvertently compromises users' privacy.

In this paper, we propose SUPOR (<u>S</u>ensitive <u>U</u>ser in<u>P</u>ut detect<u>OR</u>), a static mobile app analysis tool for detecting sensitive user inputs and identifying their as-

sociated variables in the app code as sensitive information sources. To the best of our knowledge, we are the first to study scalable detection of sensitive user inputs on smartphone platforms.

Previously, there are many existing research efforts [9, 10, 12, 23, 24, 30, 31, 34, 40] on studying the privacy related topics on predefined sensitive data sources on the phone. Our approach enables those existing efforts to be applied to sensitive user inputs as well. For example, with proper static or dynamic taint analysis, one can track the privacy disclosures of sensitive user inputs to different sinks. With static program analysis, one can also identify the vulnerabilities in the apps that may unintentionally disclosure such sensitive user inputs to public or to the attacker controlled output. One could also study how sensitive user inputs propagate to third-party advertisement libraries, etc.

In this paper, to demonstrate the usefulness of our approach, we combine SUPOR with off-the-shelf static taint analysis to detect privacy disclosures of sensitive user inputs.

The major challenges of identifying sensitive user inputs are the following:

(i) How to systematically discover the input fields from an app's UI?
(ii) How to identify which input fields are sensitive?
(iii) How to associate the sensitive input fields to the corresponding variables in the apps that store their values?

In order to detect sensitive user inputs scalably, static UI analysis is much appealing, because it is very difficult to generate test inputs to trigger all the UI screens in an app in a scalable way. For example, an app might require login, which is difficult for tools to generate desirable inputs and existing approaches usually require human intervention [26]. On the other hand, it is also extremely challenging to launch static analysis to answer the aforementioned three questions for general desktop applications.

To this end, we have studied major mobile OSes, such as Android, iOS and Windows Phone systems, and made a few important observations. Then, we implement SUPOR for Android since it is most popular.

First, we find all these mobile OSes provide a standard rapid UI development kit as part of the development framework, and most apps use such a homogeneous UI framework to develop apps. Such UI framework usually leverages a declarative language, such as XML based layout languages, to describe the UI layout, which enables us to statically discover the input fields on the UI.

Second, in order to identify which input fields are sensitive, we have to be able to render the UI, because the rendered UI screens contain important texts as hints that

guide users to enter their inputs, which can be used to identify whether the inputs are sensitive. For instance, in Figure 1, the text "User ID" describes the nature of the first input field. Statically rendering UI screens is generally very hard for arbitrary desktop applications. However, with help of WYSIWYG (What You See is What You Get) layout editing feature from the rapid UI development kits of mobile OSes, we are able to statically render the UI for most mobile apps in order to associate the descriptive text labels with the corresponding input fields. Furthermore, due to the relatively small screen size of smartphones, most text labels are concise. As such, current NLP (Natural Language processing) techniques can achieve high accuracy on identifying sensitive terms.

Third, all mobile OSes provide APIs to load the UI layouts made by rapid UI development kits and to bind with the app code. Such a binding mechanism provides us opportunities to infer the relationship between the sensitive input fields from UI layouts to the variables in the app code that store their values.

Our work makes three major contributions:

First, we devise a UI sensitiveness analysis that identifies the input fields that may accept sensitive information by leveraging UI rendering, geometrical layout analysis and NLP techniques. We modify the static rendering engine from the ADT (Android Developer Tools), so that the static rendering can be done with an APK binary instead of source code, and accurately identify the coordinates of text labels and input fields. Then, based on the insight that users typically read the text label physically close to the input field in the screen for understanding the purpose of the input field, we design an algorithm to find the optimal descriptive text label for each input field. We further leverage NLP (nature language processing) techniques [11, 22, 36] to select and map popular keywords extracted from the UIs of a massive number of apps to important sensitive categories, and use these keywords to classify the sensitive text labels and identify sensitive input fields. Our evaluation shows that SUPOR achieves an average precision of 97.3% and an average recall of 97.3% for sensitive user inputs detection.

Second, we design a context-sensitive approach to associate sensitive UI input fields to the corresponding variables in the app code. Instances of sensitive input widgets in the app code can be located using our UI analysis results in a context-insensitive fashion (i.e. based on widget IDs). We further reduce false positives by adding context-sensitivity, i.e. we leverage backward slicing and identify each input widget's residing layout by tracing back to the closest layout loading function. Only if both widget and layout identifiers match with the sensitive input field in the XML layout, we consider the widget instance is associated with the sensitive input field.

Finally, we implement a privacy disclosure detection system based on SUPOR and static taint analysis, and apply the system to 16,000 popular free Android apps collected from the Official Android Market (Google Play). The system can process 11.1 apps per minute on an eight-server cluster. Among all these apps, 355 apps are detected with sensitive user input disclosures. Our manual validation on these suspicious apps shows an overall detection accuracy of 91.3%. In addition, we conduct detailed case studies on the apps we discovered, and show interesting cases of unsafe disclosures of users' national IDs, credentials, credit card and health related information.

2 Background and Motivation Example

In this section, we provide background on sensitive user input identification.

2.1 Necessary Support for Static Sensitive User Input Identification

Modern mobile OSes, such as Android, iOS and Windows Phone system, provide frameworks and tools for rapid UI design. They usually provide a large collection of standard UI widgets, and different layouts to compose the widgets together. They also provide a declarative language, such as XML, to let the developer describe their UI designs, and further provide GUI support for WYSIWYG UI design tools. In order to design a static analysis tool for sensitive user input identification, we need four basic supporting features. The rapid UI development design in modern mobile OSes makes it feasible to achieve such features.
A: statically identify the input fields and text labels;
B: statically identify the attributes of input fields;
C: statically render the UI layout without launching the app;
D: statically map the input fields defined in the UI layouts to the app code.

These four features are necessary to statically identify the sensitive input fields on UIs. In order to infer the semantic meaning of an input field and decide whether it is sensitive, we need (*i*) the attributes of the input field; (*ii*) the surrounding descriptive text labels on the UI. Some attributes of the input fields can help us quickly understand its semantics and sensitiveness. For example, if the input type is password, we know this is a password-like input field. However, in many cases, the attributes alone are not enough to decide the semantics and sensitiveness of the input fields. In those cases, we have to rely on UI analysis. A well-designed app has to allow the user to easily identify the relevant texts for a particular input field and provide appropriate inputs based on his understanding of the meaning of texts. Based on the above observation, we need Feature C to render the UI and ob-

Table 1: UI features in different mobile OSes

	Android	iOS	Windows Phone
Layout format	XML	NIB / XIB / Storyboard	XAML/HTML
Static UI render	ADT	Xcode	Visual Studio
APIs map widgets to code	Yes	Yes	Yes

```
1  <LinearLayout android:orientation="vertical">
2    <TextView android:text="@string/tip_uid" />
3    <EditText android:id="@+id/uid" />
4    <TextView android:text="@string/tip_pwd" />
5    <EditText android:id="@+id/pwd"
            android:inputType="textPassword" />
6    <Button android:id="@+id/login"
            android:text="@string/tip_login"/>
7  </LinearLayout>
```

Figure 2: Simplified layout file *login_activity.xml*.

tain the coordinates of input fields and text labels, so that we can associate them and further reason about the sensitiveness of input fields. Once we identify the sensitive input fields, we have to find the variable in the app code used to store the values of the input field for further analysis.

We have studied Android, iOS and Windows Phone systems. As shown in Table 1, all mobile OSes provide standard formats for storing app UI layouts that we can use to achieve features A and B. All of them have IDEs that can statically render UI layouts for the WYSIWYG UI design. If we reuse this functionality we can achieve static rendering (feature C). Furthermore, all of them provide APIs for developers to map the widgets in layouts to the variables in the app code that hold the user inputs. Combined with static program analysis to understand the mapping, we will be able to achieve feature D.

2.2 Android UI Rendering

For proof of concept, the current SUPOR is designed for the Android platform. An Android app usually consists of multiple activities. Each activity provides a window to draw a UI. A UI is defined by a layout, which specifies the dimension, spacing, and placement of the content within the window. The layout consists of various interactive UI widgets (*e.g.,* input fields and buttons) as well as layout models (*e.g.,* linear or relative layout) that describe how to arrange UI widgets.

At run time, when a layout file is loaded, the Android framework parses the layout file and determines how to render the UI widgets in the window by checking the layout models and the relevant attributes of the UI widgets. At the mean time, all UI widgets in the layout are instantiated and then can be referenced in the code.

An example layout in XML is presented in Figure 2 and the code snippet of the corresponding activity is

```
1  public class LoginActivity extends Activity
       implements View.OnClickListener {
2    private EditText txtUid, txtPwd;
3    private Button btnReset;
4    protected void onCreate(Bundle bundle) {
5      super.onCreate(bundle);
6      setContentView(R.layout.login_activity);
7      txtUid = (EditText) findViewById(R.id.uid);
8      txtPwd = (EditText) findViewById(R.id.pwd);
9      btnLogin = (Button) findViewById(R.id.login);
10     btnLogin.setOnClickListener(this);
11   }
12   public void onClick(View view) {
13     String uid = txtUid.getText().toString();
14     String pwd = txtPwd.getText().toString();
15     String url = "http://www.plxx.com/Users/" +
           "login?uid=" + uid + "&pwd=" + pwd;
16     HttpClient c = new DefaultHttpClient();
17     HttpGet g = new HttpGet(url);
18     Object o = c.execute(g, new
           BasicResponseHandler());
19     // following operations are omitted
20   }
21 }
```

Figure 3: Simplified Activity example.

Figure 4: Parse tree of an example sentence.

shown in Figure 3. This layout includes five UI widgets: two text labels (`TextView`), two input fields (`EditText`) and a button. They are aligned vertically based on the `LinearLayout` at Line 1. The first text label shows "User ID" based on the attribute `android:text="@string/tip_uid"`, which indicates a string stored as a resource with the ID `tip_uid`. The *type* attribute of the second input field is `android:inputType="textPassword"`, indicating that it is designed for accepting a password, which conceals the input after the users enter it. Instead of explicitly placing text labels as in Figure 2, some developers decorate an input field with a *hint* attribute, which specifies a message that will be displayed when the input is empty. For instance, developers may choose to display "User ID" and "Password" inside the corresponding input fields using the *hint* attribute.

Figure 1 shows the rendered UI for the layout in Figure 2. The layout including all the inner widgets is loaded into the screen by calling `setContentView()` at Line 6 in Figure 3. The argument of `setContentView()` specifies the reference ID of the layout resource. Similarly, a runtime instance of a widget can also be located through a `findViewById()` call with the appropriate reference ID. For example, the reference ID `R.id.uid` is used to obtain a runtime instance of the input field at Line 7 in Figure 3.

2.3 UI Sensitiveness Analysis

Existing techniques usually consider permission protected framework APIs as the predefined sensitive data sources. However, generic framework APIs, such as `getText()`, can also obtain sensitive data from the user inputs. To precisely detect these sensitive sources,
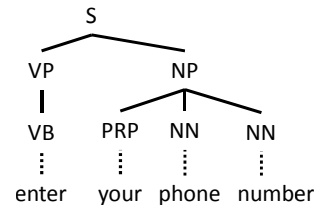
we need to determine which GUI input widgets are sensitive.

Two kinds of information are useful for this purpose. First, certain attributes of the widgets can be a good indicator about whether the input is sensitive. Using the `inputType` attribute with a value "textPassword", we can directly identify password fields. However, not all sensitive input fields use this attribute value. The hint attributes also may contain useful descriptive texts that may indicate the sensitiveness of the input fields.

Besides attributes of UI widgets, we observe that nearby text labels rendered in the UI also provide indication about the sensitiveness of the widgets. For example, a user can easily understand he is typing a user ID and a password when he sees the UI in Figure 1 because the text labels state what the input fields accept. In other words, these text labels explain the purposes of the UI widgets, and guide users to provide their inputs. Based on these observations, we propose to leverage the outcome of UI rendering to build a precise model of the UI and analyze the text labels and hints associated with the widgets to determine their sensitiveness.

The major task of analyzing text labels is to analyze the text labels' texts, which are written in natural language. As smartphones have relatively small screens, the texts shown in the UI are usually very concise and straightforward to understand. For example, these texts typically are just noun/verb phrases or short sentences (such as the ones shown in Figure 1), and tend to directly state the purposes for the corresponding GUI widgets. Since there is no need to analyze paragraphs or even long sentences, we propose a light-weight keyword-based algorithm that checks whether text labels contain any sensitive keyword to determine the sensitiveness of the corresponding GUI widgets.

2.4 Natural Language Processing

With recent research advances in the area of natural language processing (NLP), NLP techniques have been shown to be fairly accurate in highlighting grammatical structure of a natural language sentence. Recent work has also shown promising results in using NLP techniques for analyzing Android descriptions [13, 30]. In our work, we adapt NLP techniques to extract nouns and noun phrases from the texts collected from popular apps,

and identify keywords from the extracted nouns and noun phrases. We next briefly introduce the key NLP techniques used in this work.

Our approach uses **Parts Of Speech (POS) Tagging** [22,36] to identify interesting words, such as nouns, and filter unrelated words, such as conjunctives like "and/or". The technique tags a word in a sentence as corresponding to a particular part of speech (such as identifying nouns, verbs, and adjectives), based on both its definition and its relationship with adjacent and related words in a phrase, sentence, or paragraph. The state-of-the-art approaches can achieve around 97% [36] accuracy in assigning POS tags for words in well-written news articles.

Our approach uses **Phrase and Clause Parsing** to identify phrases for further inspection. Phrase and clause parsing divides a sentence into a constituent set of words (*i.e.,* phrases and clauses). These phrases and clauses logically belong together, *e.g.,* Noun Phrases and Verb Phrases. The state-of-the-art approaches can achieve around 90% [36] accuracy in identifying phrases and clauses over well-written news articles.

Our approach uses **Syntactic parsing** [21], combined with the above two techniques, to generate a parse-tree structure for a sentence, and traverse the parse tree to identify interesting phrases such as noun phrases. The parse tree of a sentence shows the hierarchical view of the syntax structure for the sentence. Figure 4 shows the parse tree for an example sentence "enter your phone number". The root node of the tree is the sentence node with the label *S*. The interior nodes of the parse tree are labeled by non-terminal categories of the grammar (*e.g.,* verb phrases *VP* and noun phrases *NP*), while the leaf nodes are labeled by terminal categories (*e.g.,* pronouns *PRP*, nouns *NN* and verbs *VB*). The tree structure provides a basis for other tasks within NLP such as question and answer, information extraction, and translation. The state of the art parsers have an F1 score of 90.4% [37].

3 Design of SUPOR

In this section, we first present our threat model, followed by an overview of SUPOR. Then, we describe each component of SUPOR in details.

3.1 Threat Model

We position SUPOR as a static UI analysis tool for detecting sensitive user inputs. Instead of focusing on malicious apps that deliberately evade detection, SUPOR is designed for efficient and scalable screening of a large number of apps. Most of the apps in the app markets are legitimate, whose developers try to monetize by gaining user popularity, even though some of them might be a little bit aggressive on exploiting user privacy for revenue. Malware can be detected by existing works [5, 15, 39],
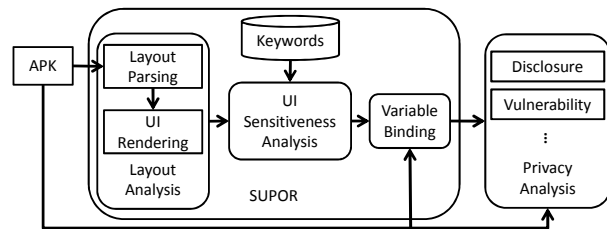

Figure 5: Overview of SUPOR.

which is out of scope of this paper.

Though the developers sometimes dynamically generate UI elements in the code other than defining the UI elements via layout files, we focus on identifying sensitive user inputs statically defined in layout files in this work.

3.2 Overview

Figure 5 shows the workflow of SUPOR. SUPOR consists of three major components: *Layout Analysis*, *UI Sensitiveness Analysis*, and *Variable Binding*. The layout analysis component accepts an APK file of an app, parses the layout files inside the APK file, and renders the layout files containing input fields. Based on the outcome of UI rendering, the UI sensitiveness analysis component associates text labels to the input fields, and determines the sensitiveness of the input fields by checking the texts in the text labels against a predefined sensitive keyword dataset (Section 3.6). The variable binding component then searches the code to identify the variables that store the values of the sensitive input fields. With variable binding, existing research efforts in studying the privacy related topics on predefined well-known sensitive data sources can be applied to sensitive user inputs. For example, one can use taint analysis to detect disclosures of sensitive user inputs or other privacy analysis to analyze vulnerabilities of sensitive user inputs in the apps. Next we describe each component in detail.

3.3 Layout Analysis

The goal of the layout analysis component is to render the UIs of an Android app, and extract the information of input fields: types, hints, and absolute coordinates, which are later used for the UI sensitiveness analysis.

As we discussed in Section 2.3, if we cannot determine the sensitiveness of an input field based on its type and hint, we need to find a text label that describes the purpose of the input field. From the *user's perspective*, the text label that describes the purpose of an input field must be *physically close to the input field* in the screen; otherwise the user may correlate the text label with other input fields and provide inappropriate inputs. Based on this insight, the layout analysis component renders the UIs as if the UIs are rendered in production runs, mimicking how users look at the UIs. Based on the rendered

```
LinearLayout @ [0, 50, 480, 752]
  → TextView @ [16, 16, 60, 33], TEXT="User ID"
  → EditText @ [16, 33, 464, 81], ID=0x7f090000
  → TextView @ [16, 81, 79, 98], TEXT="Password"
  → EditText @ [16, 98, 464, 146], ID=0x7f090001
  → Button @ [16, 146, 464, 163]
    ID=0x7f090002, TEXT="Login"
```
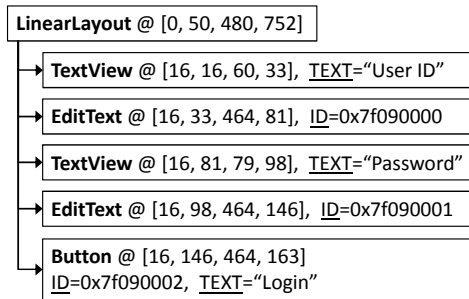
Figure 6: UI model for Figure 1 on 480x800 screen. Only the ID, relative coordinates and text of the widgets are presented here.

UIs, the distances between text labels and input fields are computed, and these distances are used later to find the best descriptive text labels for each input field. We next describe the two major steps of the layout analysis component.

The first step is to identify which layout files contain input fields by parsing the layout files in the APK of an Android app. In this work, we focus on input fields of the type `EditText` and all possible sub-types, including custom widgets in the apps. Each input field represents a potential sensitive source. However, according to our previous discussion, the sensitiveness cannot be easily determined by analyzing only the layout files. Thus, all the files containing input fields are used in the second step for UI rendering.

The second step is to obtain the coordinate information of the input fields by rendering the layout files. Using the rapid UI development kit provided by Android, the layout analysis component can effectively render standard UI widgets. For custom widgets that require more complex rendering, the layout analysis component renders them by providing the closest library superclass to obtain the best result. After rendering a layout file, the layout analysis component obtains a UI model, which is a tree-structure model where the nodes are UI widgets and the edges describe the parent-child relationship between UI widgets. Figure 6 shows the UI model obtained by rendering the layout file in Figure 2. For each rendered UI widget, the coordinates are relative to its parent container widget. Such relative coordinates cannot be directly used for measuring the distances between two UI widgets, and thus SUPOR converts the relative coordinates to absolute coordinates with regards to the screen size.

Coordinate Conversion. SUPOR computes the absolute coordinates of each UI widget level by level, starting with the root container widget. For example, in Figure 6, the root container widget is a `LinearLayout`, and its coordinates are (0, 50, 480, 752), representing the left, top, right, and bottom corners. There is

Algorithm 1 UI Widget Sensitiveness Analysis

Require: I as an input field, S as a set of text labels, KW as a pre-defined sensitive keyword dataset
Ensure: R as whether I is sensitive
1: Divide the UI plane into *nine* partitions based on I's boundary
2: **for all** $L \in S$ **do**
3:    $score = 0$
4:    **for all** $(x, y) \in L$ **do**
5:       $score \mathrel{+}= distance(I, x, y) * posWeight(I, x, y)$
6:    **end for**
7:    $L.score = score \ / \ L.numOfPixels$
8: **end for**
9: $T = min(S)$
10: $R = T.text$ matches KW

no need to convert the coordinates of the root UI widget, since its coordinates are relative to the top left corner of the screen, and thus are already absolute coordinates. For other UI widgets, SUPOR computes their absolute coordinates based on their relative coordinates and their parent container's absolute coordinates. For example, the relative coordinates of the second UI widget, `TextView`, are (16, 16, 60, 33). Since it is a child widget of the root UI widget, its absolute coordinates is computed as (16, 66, 60, 83). This process is repeated until the coordinates of every UI widget are converted.

In addition to coordinate conversion, SUPOR collects other information of the UI widgets, such as the texts in the text labels and the attributes for input fields (e.g., `ID` and `inputType`).

3.4 UI Sensitiveness Analysis

Based on the information collected from the layout analysis, the UI sensitiveness analysis component determines whether a given input field contains sensitive information. This component consists of three major steps.

First, if the input field has been assigned with certain attributes like `android:inputType="textPassword"`, it is directly considered as sensitive. With such attribute, the original inputs on the UI are concealed after users type them. In most cases these inputs are passwords.

Second, if the input field contains any hint (*i.e.*, tooltip), *e.g.*, "Enter Password Here", the words in the hint are checked: if it contains any keyword in our sensitive keyword dataset, the input field is considered sensitive; otherwise, the third step is required to determine its sensitiveness.

Third, SUPOR identifies the text label that describes the purpose of the input field, and analyzes the text in the label to determine the sensitiveness. In order to identify text labels that are close to a given input field, we provide
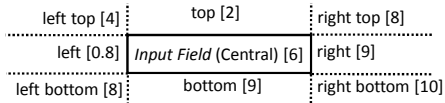
	left top [4]	top [2]	right top [8]
left [0.8]	*Input Field* (Central) [6]	right [9]
	left bottom [8]	bottom [9]	right bottom [10]

Figure 7: The partition of the UI is based on the boundary of the input field.

First Name

Last Name
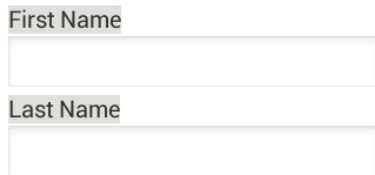
Figure 8: Example for UI widget sensitiveness analysis.

Table 2: Scores of the text labels in Figure 8.

	First Name	Last Name
1st input field	46.80	218.81
2nd input field	211.29	46.84

an algorithm to compute correlation scores for each pair of a text label and an input field based on their distances and relative positions.

The details of our algorithm is shown in Algorithm 1. At first, SUPOR divides the UI plane into nine partitions based on the boundaries of the input field. Figure 7 shows the nine partitions divided by an input field. Each text label can be placed in one or more partitions, and the input field itself is placed in the central partition. For a text label, we determine how it is correlated to an input field by computing how each pixel in a text label is correlated to the input field (Line 4). The correlation score for a pixel consists of two parts (Line 5). The first part is the Euclidean distance from the pixel to the input field, computed using the absolute coordinates. The second part is a weight based on their relative positions, *i.e.,* which of the nine partitions the widget is in. We build the position-based weight function based on our empirical observations: if the layout of the apps is top-down and left-right arranged, the text label that describes the input field is usually placed at left or on top of the input field while the left one is more likely to be the one if it exists. We assign smallest weight to the pixels in the left partition and second smallest for the top partition. The right-bottom partition is least possible so we give the largest weight to it. The detailed weights for each partition is shown in Figure 7. Based on the correlation scores of all the pixels, our algorithm uses the average of the correlation scores as the correlation score for the pair of the text label and the input field (Line 7). The label with smaller correlation score is considered more correlated to the input field.

After the correlation scores for all text labels are computed, SUPOR selects the text label that has the smallest score as the descriptive text label for the input field, and uses the pre-defined sensitive keyword dataset to determine if the label contains any sensitive keyword. If yes, the input field is considered as sensitive.

Example. Figure 8 shows an example UI that requires

Algorithm 1 for sensitiveness analysis. This example shows a UI that requests a user to enter personal information. This UI contains two input fields and two text labels. Neither can SUPOR determine the sensitiveness through their attributes, nor can SUPOR use any hint to determine the sensitiveness. SUPOR then applies Algorithm 1 on these two input fields to compute the correlation scores for each pair of text labels and input fields. The correlation scores are shown in Table 2. According to the correlation scores, SUPOR associates "First Name" to the first input field and "Last Name" to the second input field. Since our keyword dataset contains keywords "*first name*" and "*last name*" for personal information, SUPOR can declare the two input fields are sensitive.

Repeating the above steps for every input field in the app, SUPOR obtains a list of sensitive input fields. It assigns an contextual ID to each sensitive input field in the form of `<Layout_ID, Widget_ID>`, where `Layout_ID` is the ID of the layout that contains the input field and `Widget_ID` is the ID of the input field (*i.e.,* the value of the attribute "`android:id`").

3.5 Variable Binding

With the sensitive input fields identified in the previous step, the variable binding component performs context-sensitive analysis to bind the input fields to the variables in the code. The sensitive input fields are identified using contextual IDs, which include layout IDs and widget IDs. These contextual IDs can be used to directly locate input fields from the XML layout files. To find out the variables that store the values of the input fields, SUPOR leverages the binding mechanism provided by Android to load the UI layout and bind the UI widgets with the code. Such a binding mechanism enables SUPOR to associate input fields with the proper variables. We refer to these variables the *widget variables* that are bound to the input fields.

The variable binding component identifies the instances of the input fields in a context-insensitive fashion via searching the code using the APIs provided by the rapid UI development kit of Android. As shown in Section 2.2, `findViewById(ID)` is an API that loads a UI widget to the code. Its argument `ID` is the numeric ID that specifies which widget defined in the XML to load. Thus, to identify the instances of the input fields, SUPOR searches the code for such method calls, and compare their arguments to the widget IDs of the sensi-

tive input fields. If the arguments match any widget ID of the sensitive input fields, the return values of the corresponding `findViewById(ID)` are considered as the widget variables for the sensitive input fields.

One problem here is that developers may assign the same widget ID to UI widgets in different layout files, and thus different UI widgets are associated with the same numeric ID in the code. Our preliminary analysis on 5000 apps discovers that about 22% of the identified sensitive input fields have duplicate IDs within the corresponding apps. Since the context-insensitive analysis cannot distinguish the duplicate widget IDs between layout files inside an app, a lot of false positives will be presented.

To reduce false positives, SUPOR adds context-sensitivity into the analysis, associating widget variables with their corresponding layouts. Similar to loading a widget, the rapid UI development kit provides APIs to load a UI layout into the code. For example, `setContentView(ID)` with a numeric ID as the argument is used to load a UI layout to the code, as shown at Line 6 in Figure 3. Any subsequent `findViewById` with the ID `WID` as the argument returns the UI widget identified by `WID` in the newly loaded UI layout, not the UI widget identified by `WID` in the previous UI layout. Thus, to find out which layout is associated with a given widget variable, SUPOR traces back to identify the closest method call that loads a UI layout[1] along the program paths that lead to the invocation of `findViewById`. We next describe how SUPOR performs context-sensitive analysis to distinguish widget IDs between layout files. For the description below, we use `setContentView()` as an example API.

Given a widget variable, SUPOR first identifies the method call `findViewById`, and computes an interprocedural backward slice [18] of its receiver object, *i.e.,* the activity object. This backward slice traces back from `findViewById`, and includes all statements that may affect the state of the activity object. SUPOR then searches the slice backward for the method call `setContentView`, and uses the argument of the first found `setContentView` as the layout ID. For example, in Figure 3, the widget variable `txtUid` is defined by the `findViewById` at Line 7, and the activity object of this method call is an instance of `LoginActivity`. From the backward slice of the activity object, the first method call `setContentView` is found at Line 6, and thus its argument `R.layout.login_activity` is associated with `txtUid`, whose widget ID is specified by `R.id.uid`. Both `R.layout.login_activity` and `R.id.uid` can be further resolved to identify their

---

[1] SUPOR considers both `Activity.setContentView()` and `LayoutInflater.inflate()` as the methods to load UI layouts due to their prevalence.

numeric IDs, and match with the contextual IDs of sensitive input fields to determine whether `txtUid` is a widget variable for a sensitive input field.

3.6 Keyword Dataset Construction

To collect the sensitive keyword dataset, we crawl all texts in the resource files from 54,371 apps, including layout files and string resource files. We split the collected texts based on newline character (\n) to form a list of texts, and extract words from the texts to form a list of words. Both of these lists are then sorted based on the frequencies of text lines and words, respectively. We then systematically inspect these two lists with the help of the adapted NLP techniques. Next we describe how we identify sensitive keywords in detail.

First, we adapt NLP techniques to extract nouns and noun phrases from the top 5,000 frequent text lines. Our technique first uses Stanford parser [36] to parse each text line into a syntactic tree as discussed in Section 2.4, and then traverses the parse tree level by level to identify nouns and noun phrases. For the text lines that do not contain any noun or noun phrase, our technique filters out these text lines, since such text lines usually consist of only prepositions (*e.g.,* to), verbs (*e.g.,* update please), or unrecognized symbols. From the top 5,000 frequent text lines, our technique extracts 4,795 nouns and noun phrases. For the list of words, our technique filters out words that are not nouns due to the similar reasons. From the top 5,000 frequent words, our technique obtains 3,624 words. We then manually inspect these two sets of frequent nouns and noun phrases to identify sensitive keywords. As phrases other than noun phrases may indicate sensitive information, we further extract consecutive phrases consisting of two and three words from the text lists and manually inspect the top 200 frequent two-word and three-word phrases to expand our sensitive keyword set.

Second, we expand the keyword set by searching the list of text lines and the list of words using the identified words. For example, we further find "cvv code" for credit card by searching the lists using the top-ranked word "code", and find "national ID" by searching the lists using the top-ranked word "id". We also expand the keywords using synonyms of the keywords based on WordNet [11].

Third, we further expand the keywords by using Google Translate to translate the keywords from English into other languages. Currently we support Chinese and Korean besides English.

These keywords are manually classified into 10 categories, and part of the keyword dataset is presented in Table 3. Note that we do not use "Address" for the category "Personal Info". Although personal address is sensitive information, our preliminary results show that this

Table 3: Part of keyword dataset.

Category	Keywords
Credential	pin code, pin number, password
Health	weight, height, blood type, calories
Identity	username, user ID, nickname
Credit Card	credit card number, cvv code
SSN	social security number, national ID
Personal Info	first name, last name, gender, birthday
Financial Info	deposit amount, income, payment
Contact	phone number, e-mail, email, gmail
Account	log in, sign in, register
Protection	security answer, identification code

keyword also matches URL address bars in browsers, causing many false positives. Also, we do not find interesting privacy disclosures based on this keyword in our preliminary results, and thus "Address" is not used in our keyword dataset. Although this keyword dataset is not a complete dataset that covers every sensitive keyword appearing in Android apps, our evaluation results (in Section 5) show that it is a relatively complete dataset for the ten categories that we focus on in this work.

4 Implementation

In this section, we provide the details of our implementation of SUPOR, including the frameworks and tools we built upon and certain tradeoffs we make to improve the effectiveness.

SUPOR accepts APK files as inputs, and uses a tool built on top of Apktool [1] to extract resource files and bytecode from the APK file. The Dalvik bytecode is translated into an intermediate representation (IR), which is based on dexlib in Baksmali [3]. The IR is further converted to WALA [4] static single assignment format (SSA). WALA [4] works as the underlying analysis engine of SUPOR, providing various functionalities, *e.g.*, call graph building, dependency graph building, and point-to analysis.

The UI rendering engine is built on the UI rendering engine from the ADT Eclipse plug-ins Besides improving the engine to better render custom widgets, we also make the rendering more resilient using all available themes. Due to SDK version compatibility, not every layout can be rendered in every theme. We try multiple themes until we find a successful rendering. Although different themes might make UI slightly different, the effectiveness of our algorithm should not be affected. The reason is that apps should not confuse users in the successfully rendered themes, and thus our algorithm designed to mimic what users see the UIs should work accordingly.

To demonstrate the usefulness of SUPOR, we implement a privacy disclosure detection system by combining SUPOR with static taint analysis. This system enables us to conduct a study on the disclosures of sensitive user inputs. We build a taint analysis engine on top of Dalysis [24] and make several customizations to improve the effectiveness. The details of the customizations can be found at Appendix A.2.

To identify sensitive user inputs, SUPOR includes totally 11 source categories, including the 10 categories listed in Section 3.6 and an additional category *PwdLike* for the input fields identified as sensitive using their attributes such as `inputType`. The *PwdLike* category is prioritized if it has some overlapping with the other categories. Once the widget variables of the sensitive input fields are found, we consider any subsequent method calls on the variables that retrieve values from the input fields as source locations, such as `getText()`. To identify privacy disclosures of the sensitive user inputs, SUPOR mainly focuses on the information flows that transfer the sensitive data to the following two types of sinks: (1) the sinks of output channels that send the information out from the phone (*e.g.*, SMS and Network) and (2) the sinks of public places on the phone (*e.g.*, logging and content provider writes). More details are shown in Appendix A.1.

Our implementation, excluding the underlying libraries and the core taint analysis engine, accounts for about 4K source lines of code (SLoC) in Java.

5 Evaluations and Experiments

We conducted comprehensive evaluations on SUPOR over a large number of apps downloaded from the official Google Play store. We first evaluated the performance of SUPOR and demonstrated its scalability. We then measured the accuracy of the UI sensitiveness analysis and the accuracy of SUPOR in detecting disclosures of sensitive user inputs. In addition, our case studies on selected apps present practical insights of sensitive user input disclosures, which are expected to contribute to a community awareness.

5.1 Evaluation Setup

The evaluations of SUPOR were conducted on a cluster of eight servers with an Intel Xeon CPU E5-1650 and 64/128GB of RAM. During the evaluations, we launched concurrent SUPOR instances on 64-bit JVM with a maximum heap space of 16GB. On each server 3 apps were concurrently analyzed, so the cluster handled 24 apps in parallel.

In our evaluations, we used the apps collected from the official Google Play store in June 2013. We applied SUPOR to analyze 6,000 apps ranked by top downloads, with 200 apps for each category. Based on the results of the 6,000 apps, we further applied SUPOR on another 10,000 apps in 20 selected categories. Each of the 20 categories is found to have at least two apps with sensi-

Table 4: Statistics of 16,000 apps.

	#Apps	Percentage
Without Layout Files	625	3.91%
Without Input Fields	5,711	35.69%
Without Sensitive Input Fields	4,731	29.57%
With Sensitive Input Fields	4,922	30.76%
Parsing Errors	11	0.07%
TOTAL	16,000	100.00%

tive user input disclosures.

For each app, if it contains at least one input field in layout files, the app is analyzed by the UI sensitiveness analysis. If SUPOR identifies any sensitive input field of the app, the app is further analyzed by the taint analysis to detect sensitive user input disclosures. Table 4 shows the statistics of these apps. A small portion of the apps do not contain any layout files and about 1/3 of the apps do not have any input field in layout files. This is reasonable because many Game apps do not require users to enter information. 35% of the apps without layout files and 17% of the apps without input fields belong to different sub-categories of games. 11 apps (0.07%) cannot be analyzed by SUPOR due to various parsing errors in rendering their layout files. In total, 60.33% of the apps contain input fields in their layout files, among which more than half of the apps are further analyzed because sensitive input fields are found via the UI sensitiveness analysis.

As not every layout containing input fields is identified with sensitive input fields, we show the statistics of the layouts for the 4,922 apps identified with sensitive input fields. Among these apps, 47,885 layouts contain input fields and thus these layouts are rendered. Among the rendered layouts, 19,265 (40.2%) are found to contain sensitive keywords (no matter whether the keywords are associated with any input field). This is the upper bound of the number of layouts that can be identified with sensitive input fields. In fact, 17,332 (90.0%) of the 19,265 layouts with sensitive keywords are identified with sensitive input fields.

5.2 Performance Evaluation

The whole experiment for 16,000 apps takes 1439.8 minutes, making a throughput of 11.1 apps per minutes on the eight-server cluster. The following analysis is only for the 4,922 apps identified with sensitive input fields, if not specified.

The UI analysis in SUPOR includes decompiling APK files, rendering layouts, and performing UI sensitiveness analysis. For each app with sensitive input fields, SUPOR needs to perform the UI analysis for at least 1 layout and at most 190 layouts, while the median number is 7 and the average number is 9.7. Though the largest execution time required for this analysis is about 2

minutes. 96.3% of the apps require less than 10 seconds to render all layouts in an app. The median analysis time is 5.2 seconds and the average time is 5.7 seconds for one app. Compared with the other parts of SUPOR, the UI analysis is quite efficient, accounting for only 2.5% of the total analysis time on average. Also, the UI sensitiveness analysis, including the correlation score computation and keyword matching, accounts for less than 1% of the total UI analysis time, while decompiling APK files and rendering layouts take most of the time.

To detect sensitive user input disclosures, our evaluation sets a maximum analysis time of 20 minutes. 18.1% of the apps time out in our experiments but 73.7% require less than 10 minutes. The apps with many entry points tend to get stuck in taint analysis, and are more likely to timeout. Scalability of static taint analysis is a hard problem, but we are not worse than related work. The timeout mechanism is enforced for the whole analysis, but the system will wait for I/O to get partial results. In practice, we can allow a larger maximum analysis time so that more apps can be analyzed. Among the apps finished in time, the median analysis time is 1.9 minutes and the average analysis time is 3.7 minutes.

The performance results show that SUPOR is a scalable solution that can statically analyze UIs of a massive number of apps and detect sensitive user input disclosures on these apps. Compared with existing static taint analysis techniques, the static UI analysis introduced in this work is highly efficient, and its performance overhead is negligible.

5.3 Effectiveness of UI Sensitiveness Analysis

To evaluate the accuracy of the UI sensitiveness analysis, we randomly select 40 apps and manually inspect the UIs of these 40 apps to measure the accuracy of the UI sensitiveness analysis.

First, we randomly select 20 apps reported *without* sensitive input fields, and manually inspect these apps to measure the false negatives of SUPOR. In these apps, the largest number of layouts SUPOR renders is 5 and the total number of layouts containing input fields is 39 (1.95 layouts per app). SUPOR successfully renders 38 layouts and identifies 57 input fields (2.85 input fields per app). SUPOR fails to render 1 layout due to the lack of necessary themes for a third-party library. By analyzing these 57 input fields, we confirm that SUPOR has only one false negative (FN), *i.e.,* failing to mark one input field as sensitive in the app *com.strlabs.appdietas*. This input field requests users to enter their weights, belonging to the Health category in our keyword dataset. However, the text of the descriptive text label for the input field is "Peso de hoy", which is "Today Weight" in Spanish. Since our keyword dataset focuses on sensitive keywords in English, SUPOR has a false negative. Such

false negatives can be reduced by expanding our keyword dataset to support more languages.

Second, we randomly select 20 apps reported *with* sensitive input fields. Table 5 shows the detailed analysis results. Column "#Layouts " counts the number of layouts containing input fields in each app, while Column "#Layouts with Sensitive Input Fields" presents the number of layouts reported with sensitive input fields. Column "#Input Fields" lists the total number of input fields in each app and Column "#Reported Sensitive Input Fields" gives the detailed information about how many input fields are identified by checking the `inputType` attribute, by matching the hint text, and by analyzing the associated text labels. Sub-Column "Total" presents the total number of sensitive input fields identified by SUPOR in each app. Columns "FP" and "FN" show the number of false positives and the number of false negatives produced by SUPOR in classifying input fields. Column "Duplicate ID" shows if an app contains any duplicate widget ID for sensitive input fields. These duplicate IDs belong to either sensitive input fields (represented by ○) or non-sensitive input fields (●). For all the layouts in these 20 apps, SUPOR successfully renders the layouts except for App 18, which has 29 layouts containing input fields but SUPOR renders only 17 layouts. The reason is that Apktool fails to decompile the app completely.

The results show that for these 20 apps, SUPOR identifies 149 sensitive input fields with 4 FPs and 3 FNs, and thus the achieved true positives (TP) is 145. Combined with the 20 apps identified without sensitive input fields (0 FP and 1 FN), SUPOR achieves an average precision of 97.3% (precision = $\frac{TP}{TP+FP}$ = 145/149) and an average recall of 97.3% (recall = $\frac{TP}{TP+FN}$ = 145/(145+(1+3)).

We next describe the reasons for the FNs and the FPs. SUPOR has two false negatives in App 1, in which the text label "Answer" is not identified as a sensitive keyword. But according to the context, it means "security answer", which should be sensitive. Although this phrase is modeled as a sensitive phrase in our keyword dataset, SUPOR cannot easily associate "Answer" with the phrase, resulting in a false negative. In App 8, SUPOR marks an input field as sensitive because the associated text label containing the keyword "Height". However, based on the context, the app actually asks the user to enter the expected page height of a PDF file. Such issues can be alleviated by employing context-sensitive NLP analysis [19].

SUPOR also has two FPs in App 6 and App 8 due to the inaccuracy of text label association. In App 6 shown in Figure 9, the hint of the "Delivery Instructions" input field does not contain sensitive keywords, and thus SUPOR identifies the close text label for determining its sensitivity. However, SUPOR incorrectly asso-
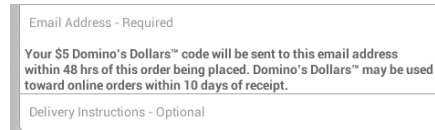

Figure 9: False positive example in UI sensitiveness analysis.

ciates a description label of "Email" to the "Delivery Instructions" input field based on their close distances. Since this description contains sensitive keywords such as email, SUPOR considers the "Delivery Instructions" input field as sensitive, causing a false positive. Finally, SUPOR has both FPs and FNs for App 14, since its arrangements of input fields and their text labels are not accurately captured by our position-based weights that give preferences for left and top positioned text labels.

To evaluate the effectiveness of resolving duplicate IDs, We instrumented SUPOR to output detailed information when identifying the widget variables. We did not find any case where SUPOR incorrectly associates the widget variables with the input fields based on the contextual IDs, but potentially SUPOR may have inaccurate results due to infeasible sequences of entry points that can be executed. We next present an example to show how backward slicing help SUPOR distinguish duplicate widget IDs. App 17 has two layouts with the same hierarchy. Layout A contains a sensitive input field with the ID $w1$ while Layout B contains a non-sensitive input field with the same ID $w1$. Both layouts are loaded via `LayoutInflater.inflate` and then `findViewById` is invoked separately to obtain the enclosed input fields. Without the backward slicing, SUPOR considers the input field with the ID $w1$ in the Layout B as sensitive, which is a false positive. With the backward slicing, SUPOR can distinguish the input field with the ID $w1$ in Layout B with the input field with the ID $w1$ in Layout A, and correctly filter out the non-sensitive input field in Layout B.

5.4 Accuracy of Detecting Sensitive User Input Disclosures

In our experiments, 355 apps are reported with sensitive user input disclosures. The reported apps belong to 25 out of the 30 categories in Google Play Store and 20 categories have at least 2 apps reported. We next report the accuracy of detecting sensitive user input disclosures.

Figure 10 shows the number of true positives and the number of false positives by taint source and sink categories. If an app is reported with multiple disclosure flows and one of them is a false positive, the app is considered as a false positive. Through manually evaluating the 104 apps reported cases from the first 6,000 analyzed apps, we find false positives in 9 apps. Therefore, the overall false positive rate is about 8.7%, *i.e.,* the accu-

Table 5: UI analysis details for 20 randomly chosen apps.

App ID	#Layouts	#Input Fields	#Layouts with Sensitive Input Fields	#Reported Sensitive Input Fields				FP	FN	Duplicate ID
				Password	Hint	Label	Total			
1	8	18	4	6	0	3	9		2	○
2	37	77	2	0	0	8	8			
3	3	3	1	0	1	0	1			
4	4	9	3	0	0	6	6			○
5	5	7	1	1	0	0	1			
6	17	52	10	6	12	12	30	1		○
7	4	5	2	0	0	3	3			
8	15	22	9	8	3	2	13	1		
9	3	7	1	1	1	0	2			
10	7	16	1	0	0	1	1			
11	5	6	1	1	1	0	2			
12	17	33	8	8	9	0	17			○ ●
13	26	60	10	0	0	12	12			○ ●
14	2	8	2	1	0	4	5	2	1	
15	14	26	5	2	3	0	5			○ ●
16	4	7	1	1	0	0	1			
17	4	8	3	2	3	0	5			●
18	29	25	4	4	0	6	10			○
19	24	37	8	9	6	1	16			○
20	1	2	1	0	2	0	2			
Total	229	428	77	50	41	58	149	4	3	
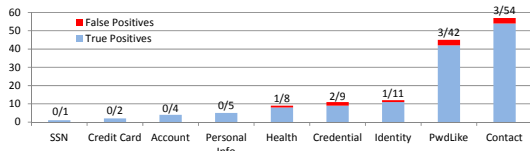
(a) TPs and FPs by source categories.
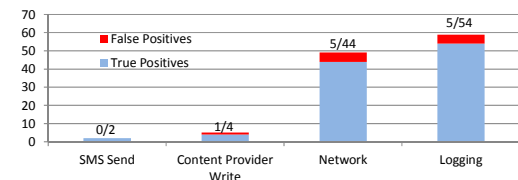
(b) TPs and FPs by sink categories.

Figure 10: True positives and false positives by source/sink categories for the reported apps.

racy of privacy disclosure detection is 91.3%. We investigated the false positives and found that these false positives were mostly resulted from the limitations of the underlying taint analysis framework, such as the lack of accurate modeling of arrays.

5.5 Case Studies

To improve the community's awareness and understanding of sensitive user input disclosures, we conducted cases studies on four selected apps from the source categories SSN, PwdLike, Credit Card, and Health. These case studies present interesting facts of sensitive user input disclosures, and also demonstrate the usefulness of SUPOR. We also inform the developers of the apps mentioned in this section about the detected disclosures.

com.yes123.mobile is an app for job hunting. The users are required to register with their national ID and a password to use the service. When the users input the ID and password, and then click log in[2], the app sends both their national IDs and passwords via Internet without any protection (*e.g.,* hashing or through HTTPS channel). Since national ID is quite sensitive (similar as Social Security Number), such limited protection in transmission may lead to serious privacy disclosure problems.

The second example app (*craigs.pro.plus*) shows a legitimate disclosure that uses HTTPS connections to send user sensitive inputs to its server for authentication. Even though the password itself is not encoded (*e.g.,* hashing), we believe HTTPS connections provide a better protection layer to resist the disclosures during communications. Also we find that popular apps developed by enterprise companies are more likely to adopt HTTPS, providing better protection for their users.

To better understand whether sensitive user inputs are properly protected, we further inspect 104 apps, of which 44 apps send sensitive user inputs via network. Among these 44 apps, only 10 of them adopt HTTPS connections, while the majority of apps transmit sensitive user inputs in plain text via HTTP connections. Such study

---

[2]The UI is shown in Figure 12 in Appendix B.1.

Figure 11: Case study: credit card information disclosure example.

results indicate that most developers are still unaware of the risks posed by sensitive user input disclosures, and more efforts should be devoted to provide more protections on sensitive user inputs.

Our last example app (*com.nitrogen.android*) discloses credit card information, a critical financial information provided by the users. Figure 11 shows the rendered UI of the app. The three input fields record credit card number, credit card security number, and the card holder's name. Because these fields are not decorated with `textPassword` input type and they do not contain any hints, SUPOR uses the UI sensitiveness analysis to compute correlation scores for each text label. As we can see from the UI, the text label "Credit Card Number" and the text label "Credit Card Security Number" are equally close to the first input field. As our algorithm considers weights based on the relative positions between text labels and input fields, SUPOR correctly associates the corresponding text labels for these three input fields, and the taint analysis identifies sensitive user input disclosures for all these three input fields to logging. SUPOR also identifies apps that disclose personal health information to logging, and the example app is shown in Appendix B.2.

Although Google tries to get rid of some of the known sinks that contribute most of the public leaks by releasing new Android versions, many people globally may still continue using older Android releases for a very long time (about 14.2% of Android phones globally using versions older than Jelly Bean [2]). If malware accesses the logs on these devices, all the credit card information can be exploited to malicious adversaries. Thus, certain level of protection is necessary for older versions of apps. Also, SUPOR finds that some apps actually sanitize the sensitive user inputs (*e.g.,* hashing) before these inputs are disclosed in public places on the phone, indicating that a portion of developers do pay attention to protecting sensitive user input disclosures on the phone.

6 Discussion

SUPOR is designed as an effective and scalable solution to screening a large number of apps for sensitive user inputs. In this work, we have demonstrated that SUPOR can be combined with static taint analysis to automatically detect potential sensitive user input disclosures. Such analysis can be directly employed by app markets to raise warnings, or by developers to verify whether their apps accidentally disclose sensitive user inputs. Also, SUPOR can be paired with dynamic taint analysis to alert users before the sensitive user inputs escape from the phones.

SUPOR focuses on input fields, a major type of UI widgets to collect user inputs. Such UI widgets record what user type and contain high entropy, unlike yes/no buttons which contain low entropy. It is quite straightforward to extend our current approach to handle more diverse widgets.

SUPOR chooses the light-weight keyword-based technique to determine the sensitiveness of input fields since the texts contained in the associated text labels are usually short and straightforward to understand. Our evaluations show that in general these keywords are highly effective in determining the sensitiveness of input fields. Certain keywords may produce false positives since these keywords have different meanings under different contexts. To alleviate such issues, we may leverage more advanced NLP techniques that consider contexts [19].

7 Related Work

Many great research works [6, 8–10, 12, 14, 16, 23, 24, 34, 38] focus on privacy leakage problems on predefined sensitive data sources on the phone. SUPOR identifies sensitive user inputs, and may enable most of the existing research on privacy studies to be applied to sensitive user inputs. As a result, our research compliments the existing works. FlowDroid [6] also employs a limited form of sensitive input fields—password fields. Compared with FlowDroid, we leverage static UI rendering and NLP techniques to identify different categories of sensitive input fields in an extensible manner. Susi [33] employs a machine learning approach to detect pre-defined source/sinks from Android Framework. In contrast, SUPOR focus on a totally different type of sensitive sources–user inputs through GUI.

Moreover, a few approaches are designed for controlling the known privacy leaks. AppFence [17] employs fake data or network blocking to protect privacy leaks to Internet with user supplied policies. Nadkarni *et al.*provide new OS mechanisms for proper information sharing cross apps [28].

NLP techniques have been used to study app descriptions [13, 30, 31]. WHYPER [30] and AutoCog [31] leverages NLP techniques to understand whether the application descriptions reflect the permission usage. CHABADA [13] also applies topic modelling, an NLP

technique to detecting malicious behaviors of Android apps. It generates clusters according to the topic, which consists of a cluster of words that frequently occur together. Then, it tries to detect the outliers as malicious behaviors. CHABADA does not focus on detecting privacy leaks. On the other hand, SUPOR leverages NLP techniques to identify sensitive keywords and further use those keywords to classify the descriptive text labels and the associated input fields.

Furthermore, there are a few important related works using UI related information to detect different types of vulnerabilities and attacks. AsDroid [20] checks UI text to detect the contradiction between expected behavior inferred from the UI and the program behavior represented by APIs. Chen *et al.* study the GUI spoofing vulnerabilities in IE browser [7]. Mulliner *et al.* discover GUI element misuse (GEM), a type of GUI related access control violation vulnerabilities and design GEM Miner to automatically detect GEMs [27]. SUPOR focuses on sensitive user input identification which is different from the problems studied by these existing works.

The closest related work is UIPicker [29], which also focuses on sensitive user input identification. UIPicker uses supervised learning to train a classifier based on the features extracted from the texts and the layout descriptions of the UI elements. It also considers the texts of the sibling elements in the layout file. Unlike UIPicker that uses sibling elements in the layout file as the description text for a UI widget, which could easily include unrelated texts as features, SUPOR selects only the text labels that are physically close to input fields in the screen, mimicking how users look at the UI, and uses the texts in the text labels to determine the sensitiveness of the input fields. Also, their techniques in extracting privacy-related texts could complement our NLP techniques to further improve our keyword dataset construction.

In the software engineering domain, there are quite a few efforts on GUI reverse engineering [25, 26, 32, 35] for GUI testing. GUITAR is a well-known framework for general GUI testing, and GUI ripper [26], a component of GUITAR targets general desktop applications, uses dynamic analysis to extract GUI related information and requires human intervention when the tools cannot fill in proper information in the applications. In [25] and [32], two different approaches have been proposed to convert the hard-coded GUI layout to model-based layout (such as XML/HTML layout). GUISurfer leverages source code to derive the relationships between different given UI widgets. In contract, SUPOR focuses on mobile apps and in particular Android apps, and leverages the facility from existing rapid UI development kits to identify and render UI widgets statically.

8 Conclusions

In this paper, we study the possibility of scalably detecting sensitive user inputs, an important yet mostly neglected sensitive source in mobile apps. We leverage the rapid UI development kits of modern mobile OSes to detect sensitive input fields and correlate these input fields to the app code, enabling various privacy analyses on sensitive user inputs. We design and implement SUPOR, a new static analysis tool that automatically identifies sensitive input fields by analyzing both input field attributes and surrounding descriptive text labels through static UI parsing and rendering. Leveraging NLP techniques, we build mobile app specific sensitive word vocabularies that can be used to determine the sensitiveness of given texts. To enable various privacy analyses on sensitive user inputs, we further propose a context-sensitive approach to associate the input fields with corresponding variables in the app code.

To demonstrate the usefulness of SUPOR, we build a privacy disclosure discovery system by combining SUPOR with static taint analysis to analyze the sensitive information of the variables that store the user inputs from the identified sensitive input fields. We apply the system to 16,000 popular Android apps, and SUPOR achieves an average precision of 97.3% and also an average recall of 97.3% in detecting sensitive user inputs. SUPOR finds 355 apps with privacy disclosures and the false positive rate is 8.7%. We also demonstrate interesting real-world cases related to national ID, username/password, credit card and health information.

9 Acknowledgements

References

[1] Android-ApkTool: A tool for reverse engineering Android apk file. `https://code.google.com/p/android-apktool`.

[2] Android Dashboards. `https://developer.android.com/about/dashboards/index.html`. Accessed: 20 Feb 2015.

[3] Baksmali: a disassembler for Android's dex format. `https://code.google.com/p/smali`.

[4] WALA: T.J. Watson Libraries for Analysis. `http://wala.sourceforge.net`.

[5] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., AND RIECK, K. DREBIN: Effective and explainable detection of Android malware in your pocket. In *NDSS* (2014).

[6] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI* (2014).

[7] CHEN, S., MESEGUER, J., SASSE, R., WANG, H. J., AND WANG, Y.-M. A systematic approach to uncover security flaws in GUI logic. In *S&P (Oakland)* (2007).

[8] EGELE, M., KRUEGEL, C., KIRDA, E., AND VIGNA, G. PiOS: Detecting privacy leaks in iOS applications. In *NDSS* (2011).

[9] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010).

[10] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of Android application security. In *USENIX Security* (2011).

[11] FELLBAUM, C., Ed. *WordNet An Electronic Lexical Database*. The MIT Press, 1998.

[12] GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST* (2012).

[13] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *ICSE* (2014).

[14] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic detection of capability leaks in stock Android smartphones. In *NDSS* (2012).

[15] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. Riskranker: Scalable and accurate zero-day Android malware detection. In *MobiSys* (2012).

[16] HAN, J., YAN, Q., GAO, D., ZHOU, J., AND DENG, R. Comparing mobile privacy protection through cross-platform applications. In *NDSS* (2013).

[17] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *CCS* (2011).

[18] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *SIGPLAN Not. 23*, 7 (June 1988).

[19] HUANG, E. H., SOCHER, R., MANNING, C. D., AND NG, A. Y. Improving word representations via global context and multiple word prototypes. In *ACL* (2012).

[20] HUANG, J., ZHANG, X., TAN, L., WANG, P., AND LIANG, B. Asdroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *ICSE* (2014).

[21] JURAFSKY, D., AND MARTIN, J. H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[22] KLEIN, D., AND MANNING, C. D. Accurate unlexicalized parsing. In *ACL* (2003).

[23] LU, K., LI, Z., KEMERLIS, V., WU, Z., LU, L., ZHENG, C., QIAN, Z., LEE, W., AND JIANG, G. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS* (2015).

[24] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS* (2012).

[25] LUTTEROTH, C. Automated reverse engineering of hard-coded GUI layouts. In *AUIC* (2008).

Figure 12: Case study: national ID and password disclosure example without protection.

[26] MEMON, A., BANERJEE, I., AND NAGARAJAN, A. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE* (2003).

[27] MULLINER, C., ROBERTSON, W., AND KIRDA, E. Hidden GEMs: Automated discovery of access control vulnerabilities in graphical user interfaces. In *S&P (Oakland)* (2014).

[28] NADKARNI, A., AND ENCK, W. Preventing accidental data disclosure in modern operating systems. In *CCS* (2013).

[29] NAN, Y., YANG, M., YANG, Z., ZHOU, S., GU, G., AND WANG, X. UIPicker: User-input privacy identification in mobile applications. In *USENIX Security* (2015).

[30] PANDITA, R., XIAO, X., YANG, W., ENCK, W., AND XIE, T. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security* (2013).

[31] QU, Z., RASTOGI, V., ZHANG, X., CHEN, Y., ZHU, T., AND CHEN, Z. AutoCog: Measuring the description-to-permission fidelity in Android applications. In *CCS* (2014).

[32] RAMÓN, Ó. S., CUADRADO, J. S., AND MOLINA, J. G. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering 21*, 2 (2014).

[33] RASTHOFER, S., ARZT, S., AND BODDEN, E. A machine-learning approach for classifying and categorizing Android sources and sinks. In *NDSS* (2014).

[34] RASTOGI, V., CHEN, Y., AND ENCK, W. AppsPlayground: Automatic security analysis of smartphone applications. In *ASI-ACCS* (2013).

[35] SILVA, J. C., SILVA, C., GONÇALO, R. D., SARAIVA, J., AND CAMPOS, J. C. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In *EICS* (2010).

[36] The Stanford Natural Language Processing Group, 1999. http://nlp.stanford.edu/.

[37] SOCHER, R., BAUER, J., MANNING, C. D., AND NG, A. Y. Parsing with compositional vector grammars. In *ACL* (2013).

[38] YANG, Z., YANG, M., ZHANG, Y., GU, G., NING, P., AND WANG, X. S. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *CCS* (2013).

[39] ZHOU, Y., AND JIANG, X. Dissecting Android malware: Characterization and evolution. In *S&P (Oakland)* (2012).

[40] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in Android applications. In *NDSS* (2013).

Appendix

A Taint Analysis

The details of sinks and customizations of the taint analysi engine are shown in this section.

A.1 Sink Dataset

The sink dataset includes five categories of sink APIs, among which two categories are SMS send (*e.g.,*`SmsManager.sendTextMessage()`) and Network (*e.g.,*`HttpClient.execute()`). The other three are related to local storage: logging (*e.g.,*`Log.d()`), content provider writes (*e.g.,*`ContentResolver.insert()`), and local file writes (*e.g.,*`OutputStream.write()`). Totally there are 236 APIs.

A.2 Customizations of Taint Analysis

Our taint analysis engine constraints the taint propagation to only variables and method-call returns of `String` type. Therefore, method calls that return primitive types (*e.g.,*`int`) are ignored. There are two major reasons for making this tradeoff. The first is that the sensitive information categories we focus on are passwords, user names, emails, and so on, and these are usually not numeric values. The second is that empirically we found a quite number of false positives related to flows of primitive types due to the incompleteness of API models for the Android framework. This observation-based refinement suppresses many false positives. For example, one false warning we observed is that the length of a tainted string (`tainted.length()`) is logged, and tracking such length causes too many false positives afterwards. Since such flow does not disclose significant information of the user inputs, removing the tracking of such primitive values reduces the sources to track and improves the precision of the tracking.

To further suppress false warnings, we model data structures of key-value pairs, such as `Bundle` and `BasicNameValuePair`. `Bundle` is widely used for storing an activity's previously frozen state, and `BasicNameValuePair` is usually used to encode name-value pairs for HTTP URL parameters or other web transmission parameters, such as JSON. For each detected disclosure flow, we record the keys when the analysis finds method calls that insert values into the data structures, *e.g.,*`bundle.put("key1", tainted)`. For any subsequent method call that retrieves values from the data structures, *e.g.,*`bundle.get("key2")`, we compare the key for retrieving values `key2` with the recorded keys. If no matches are found, we filter out the disclosure flow.

B Example Apps in Case Studies

B.1 Example App for Disclosing National IDs

The UI for the first example app described in Section 5.5, *com.yes123.mobile*, is shown in Figure 12.

Figure 13: Case study: health information disclosure.

B.2 Example App for Disclosing Health Information

Figure 13 shows the rendered UI of the layout *dpacacl* in app *com.canofsleep.wwdiary*, which belongs to the category HEALTH && FITNESS. This app discloses personal health information through the user inputs collected from the UI. As we can see, even though all input fields on the UI hold hint texts, these texts do not contain any sensitive keywords. Therefore, SUPOR still needs to identify the best descriptive text label for each input field. Based on the UI sensitiveness analysis, SUPOR successfully marks the first three input fields as sensitive, *i.e.,* the input fields that accept *weight*, *height* and *age*. But based on the taint analysis, only the first two input fields are detected with disclosure flows to logging. Similar to financial information, such health information about users' wellness is also very sensitive to the users.