STATIC ANALYSIS OF ANDROID APPS WITH TEXT ANALYSIS AND BI-DIRECTIONAL PROPAGATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Jianjun Huang

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2017

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

This work is dedicated to my wife Jie Zhang.

水过竞千帆，遥遥汪洋叹。
屈指昼夜轮，多少日月转。
文章千古事，史载三两篇。
回首明镜里，仍是旧时颜。
我取昆仑胆，谢氏堂前燕。
张眼睥天下，捷书先流传。

ACKNOWLEDGMENTS

First and foremost, I would like to express my special appreciation and deepest gratitude to my advisor, Professor Xiangyu Zhang. I would like to thank him for his extraordinary guidance, encouragement and patience during my whole PhD study. He provided me the freedom to work on topics I was interested in. When I got stuck, he was always there to help and made sure I was headed in the right direction. He stayed up late at night with me for submission deadlines. He polished all my writings and provided me valuable suggestions for improving my professional writing skills. He took the time from his busy schedule and helped me to get prepared for presentations and job talks. I am so fortunate to have him as my advisor.

I would like to sincerely thank Professor Lin Tan. She guided me in the research of text analysis. I would also like to thank Professor Ninghui Li and Professor Vernon Rego for their time and efforts to serve on my PhD committee. I thank them for all of their valuable inputs and suggestions. I also thank Professor Tiark Rompf, Professor Hubert Dunsmore, and Professor Bharat Bhargava for serving on my qualifying/preliminary exam committee and their suggestions.

I would like to thank Dr. Zhichun Li and Dr. Zhenyu Wu in NEC Labs, Dr. Xusheng Xiao in Case Western Reserve University, and Dr. Chen Tian in Huawei for their helps during my intern and efforts on my research and paper writing.

I would like to extend my thanks to the research group members, Dr. Yunhui Zheng, Dr. Tao Bao, Dr. Peng Liu, Dr. Yousra Aafer, Yonghwi Kwon, Weihang Wang and Shiqing Ma for their kind discussions and helps in both research and real life.

Finally, I would also thank my parents, Shaoyan Huang and Maoxiu Li, and my wife, Jie Zhang, for their love, support and encouragement. Without them, this dissertation could not have happened.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

API	Application Program Interface
JSON	JavaScript Object Notation
UI	User Interface
GUI	Graphical User Interface
URL	Uniform Resource Locator
WALA	T. J. Watson Libraries for Analysis
APK	Android Package Kit
DEX	Dalvik Executable Format
ICC	Inter-Component Communication
SDK	Software Development Kit
ADT	Android Development Tools
SMS	Short Message Service
IR	Intermediate Representation
SSA	Static Single Assignment
IMEI	International Mobile Equipment Identity
NLP	Natural Language Processing
APP	(Android) Application
WYSIWYG	What You See Is What You Get
IDE	Integrated Development Environment
LHS	Left Hand Side
RHS	Right Hand Side
FP	False Positive
TP	True Positive
FN	False Negative

ABSTRACT

Huang, Jianjun PhD, Purdue University, December 2017. Static Analysis of Android Apps with Text Analysis and Bi-directional Propagation. Major Professor: Xiangyu Zhang.

While smartphones and mobile apps have been an integral part of our life, personal security issues on smartphones become a serious concern. Privacy leakage, namely sensitive data disclosures, happens frequently in mobile apps to disclose the user's sensitive information to untrusted, even malicious, third-party service providers, leading to serious problems. Besides, stealthy behaviors that are performed without the user's acknowledgment may cause unexpected phone charges or leakage of sensitive information.

To address these problems, many approaches have been proposed. However, previous mobile privacy related research efforts have largely focused on predefined known sources managed by smartphones. More specifically, they focus on the API functions that directly return sensitive values. Some other information sources, such as the user inputs through user interface and data obtained from network or files, have been mostly neglected, even though such sources may contain a lot of sensitive information. In addition, the research efforts on detecting stealthy behaviors also depend on identifying suspicious behaviors with known actions, e.g., known premium phone numbers or URLs of malicious websites.

In this dissertation, we present two automated techniques for the purpose of comprehensively sensitive data disclosure detection. Moreover, we propose a novel technique to detect stealthy behaviors in Android apps.

Firstly, we examine the possibility of scalably detecting sensitive user inputs from mobile apps. We design and implement SUPOR, a novel static analysis tool that automatically examines the user interface to identify sensitive user inputs containing critical user data, such as user credentials, finance and medical data. SUPOR mimics from the user's perspective to associate input fields in user interfaces with most correlated text labels and utilizes text analysis to determine the sensitiveness of the user inputs. With the knowledge of sensitive user inputs, we are then able to detect their disclosures with the help of taint analysis.

Secondly, we develop BIDTEXT to address the issues of detecting sensitive data disclosures where the data is generated by generic API functions whose return values cannot be easily recognized as sensitive or insensitive. BIDTEXT leverages the context of the data, associates the correlated text labels to corresponding variables and then applies text analysis to determine the sensitiveness of the data held by the variables. The intuition here is that the context of programs contains useful information to indicate what the variables may hold. BIDTEXT also features a novel bi-directional propagation technique through forward and backward data-flow to enhance static sensitive data disclosure detection.

Thirdly, we develop AsDroid to detect stealthy behaviors in Android apps by checking the contradiction between user expectation, which is represented by user interface, and program behavior that can be abstracted by API invocations. We model API invocations with different types of intents and backwardly propagate the intents to top level functions, e.g., a user interaction function. We then analyze the text extracted from the user interface component associated with the top level function. Semantic mismatch of the two indicates stealthy behavior.

To sum up, in this dissertation, we present SUPOR to detect sensitive user inputs, and BIDTEXT to determine the sensitiveness of the data generated by generic API functions. We also propose bi-directional propagation to enhance sensitive data disclosure detection. In addition, we inspect the contradiction between program behaviors and user expectations to detect stealthy behaviors in Android apps.

1 INTRODUCTION

While smartphones and mobile apps have been an essential part of our life, personal security issues on smartphones, including privacy and malicious behaviors, become a serious concern.

Privacy Issues. Previous mobile privacy related research efforts have largely focused on predefined known sources managed by smartphones. More specifically, they focus on the API functions that return sensitive values, 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 is the *sensitive user input*, which refers to the sensitive information entered by users via the user interface. Many mobile apps today acquire sensitive credentials, financial, health, and medical information from users through the user interfaces. While all kinds of user inputs, either sensitive or insensitive are retrieved in the code via the same API functions, traditional sensitive data disclosure detection techniques that require predefined data sources are not enough because they lack mechanism of deciding the sensitiveness of those API returns. In the context of static detection of mobile apps, prior to performing sensitive data disclosure detection, we must resolve the challenges of discovering the input fields from an app's user interface, identifying which input fields are sensitive and associating the sensitive input fields to the corresponding variables in the code.

On the other hand, existing techniques require the data sensitiveness of the sources to be known before the static detection is conducted. Even the above problem is related to determine the data sensitiveness based on where the data is produced. With the knowledge of sensitiveness at the source points, a forward data flow needs to be observed between sources and sinks in order to report a disclosure defect. However, some generic API functions may return sensitive values, depending on the context, although they may return insensitive values in many cases, and we have no way to understand the data sensitiveness from where the data is generated. For example, a local file or the network response may contain sensitive values but given the file or the network request, we cannot claim the sensitiveness of the response. In such cases, most existing approaches would not work properly. We cannot simply treat the generic API functions as the sensitive data sources as that will lead to a large number of false warnings, or just ignore all of such cases because we can expect missing warnings. In addition, forward data flow analysis is insufficient. In many cases, a piece of data may be first emitted through a sink and then later recognized as sensitive somehow.

Stealthy Behaviors. Detecting malicious behaviors, especially stealthy behaviors, are also a hot research topic. Existing techniques mainly depends on identifying certain known fingerprints of the malicious operations in Android apps. For example, if an app contains directly making phone calls or sending short messages to a known premium numbers, or accessing a URL of known malicious website, the app will be reported as malware. More sophisticated cases cannot be detected. For instance, adversaries may obfuscate the numbers or URLs in the code such that static analysis is not able to understand whether the numbers or URLs are blacklisted. Besides, maintaining a blacklist of the numbers and URLs are not trivial. In some countries, the premium numbers are the same as normal phone numbers. They can also be actively changed to avoid the blacklist detection. Since the code behaviors, represented by API invocations, of malicious actions and benign operations are the same, we cannot report malicious behaviors by only inspecting the API calls when we do not know whether the destinations are good or bad.

1.1 Thesis Statement

This dissertation addresses the important issue on the detection of sensitive data disclosures in mobile apps by presenting two approaches. First, it proposes SUPOR, a static technique that can automatically, scalably and precisely detect the data sensitiveness of user inputs from user interfaces. Second, it introduces BIDTEXT, a technique that can recognize the sensitiveness of data generated from even more generic API functions, such as data from network or files. The dissertation also presents AsDroid to detect stealthy behaviors in Android apps.

The thesis statement is as follows: Existing privacy related techniques that have mostly focused on predefined sensitive data sources are not enough to detect sensitive data disclosures, when the data generated by some generic API functions (such as reading data from user interface, files, network and so on) are neglected; and utilizing maliciously known destinations to detect stealthy behaviors is not sufficient. Proposing an automated technique to identify the sensitiveness of user inputs can help the detectors discover the sensitive user input disclosure problems. Introducing the bi-directional text correlation analysis can handle even more cases in which the data sensitiveness cannot be determined at the data generation points. Combining code behavior analysis and user interface analysis can tell whether certain program behaviors contradict the user expectation, namely, whether they are stealthy behaviors.

1.2 Contributions

The contributions of this dissertation are as follows:

- We propose SUPOR, a static technique that can automatically, scalably and precisely determine the data sensitiveness of user inputs in Android apps. SUPOR achieves the following three challenging goals: (1) it systematically discovers the input fields from an app's UI; (2) identifies which input fields are sensitive by associating the input fields with mostly correlated text labels and performing text analysis to determine whether the user inputs contain sensitive data; (3) and associates the sensitive input fields to the corresponding variables in the apps that store their values.

- We design and develop a novel technique, BIDTEXT, to statically detect sensitive data disclosures in Android apps while the data can be generated from more generic API functions like reading from files or network. Since the data sensitiveness of such generic API functions cannot be determined from the definition locations, we address the challenges by typing the correlated text labels to corresponding variables and propagating them bi-directionally along forward and backward data-flow. The problem is formalized in a type system and we have obtained some preliminary results for the prototype we implement.

- We present AsDroid to statically detect stealthy behaviors in Android apps without the knowledge of maliciously destinations like premium numbers, malicious URLs. We resolve the problem by inspecting the contradiction between user expectation and program behaviors. The former one can be abstracted from the user interface and the latter one is represented by API invocations. We propagate the intents of API calls to top level functions and then check if they mismatch with what the associated user interface indicates. The analysis is formalized by datalog rules.

1.3 Dissertation Organization

This dissertation is organized as follows:

Chapter 2 discusses the design, implementation and evaluation of SUPOR. Chapter 3 details the problem of detecting sensitive data disclosures for generic data and how we solve the problem through bi-directional text correlation analysis. Chapter 4 talks about the motivation, design and evaluation of AsDroid to detect stealthy behaviors.

2 SUPOR: PRECISE AND SCALABLE SENSITIVE USER INPUT DETECTION FOR ANDROID APPLICATIONS

While smartphones and mobile apps have been an essential part of our life, 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 section, 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.

2.1 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 sen-

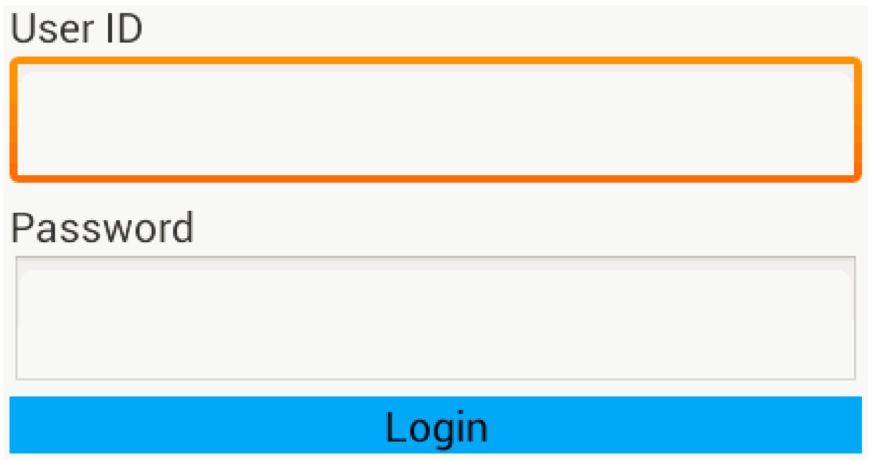


Figure 2.1.: Example sensitive user inputs.

sitive and private personal data. Despite the capabilities to meet users' needs, data privacy in smartphones becomes a major concern.

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 2.1 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 section, we propose SUPOR (Sensitive User inPut detectOR), a static mobile app analysis tool for detecting sensitive user inputs and identifying their associated 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 [1–9] 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.

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 [10]. 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 2.1, 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 [11–14] 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.

2.2 Background and Motivation Example

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

2.2.1 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 obtain 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 2.1, 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

Table 2.1.: 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

```
<LinearLayout android:orientation="vertical">
  <TextView android:text="@string/tip_uid" />
  <EditText android:id="@+id/uid" />
  <TextView android:text="@string/tip_pwd" />
  <EditText android:id="@+id/pwd"
      android:inputType="textPassword" />
  <Button android:id="@+id/login"
      android:text="@string/tip_login"/>
</LinearLayout>
```

Figure 2.2.: Simplified layout file *login_activity.xml*.

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.

2.2.2 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

```java
public class LoginActivity extends Activity implements
   View.OnClickListener {
  private EditText txtUid, txtPwd;
  private Button btnReset;
  protected void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView(R.layout.login_activity);
    txtUid = (EditText) findViewById(R.id.uid);
    txtPwd = (EditText) findViewById(R.id.pwd);
    btnLogin = (Button) findViewById(R.id.login);
    btnLogin.setOnClickListener(this);
  }
  public void onClick(View view) {
    String uid = txtUid.getText().toString();
    String pwd = txtPwd.getText().toString();
    String url = "http://www.plxx.com/Users/" + "login?uid=" +
       uid + "&pwd=" + pwd;
    HttpClient c = new DefaultHttpClient();
    HttpGet g = new HttpGet(url);
    Object o = c.execute(g, new BasicResponseHandler());
    // following operations are omitted
  }
}
```

Figure 2.3.: Simplified activity example.

(*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 2.2 and the code snippet of the corresponding activity is shown in Figure 2.3. 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 2.2, 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 2.1 shows the rendered UI for the layout in Figure 2.2. The layout including all the inner widgets is loaded into the screen by calling `setContentView()` at Line 6 in Figure 2.3. 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 2.3.

2.2.3 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`

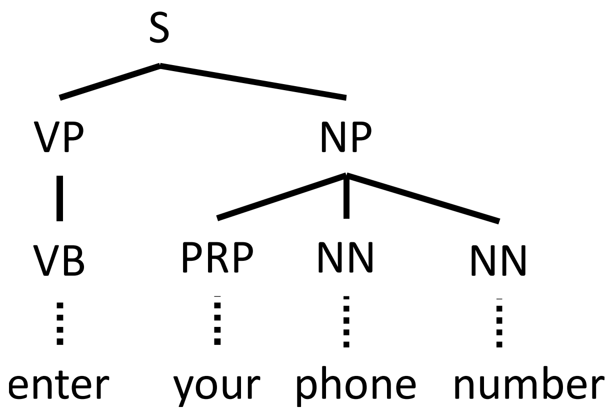


Figure 2.4.: Parse tree of an example sentence.

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 2.1 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 2.1), 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.

2.2.4 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 [7, 15]. 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** [11, 12] 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% [12] 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% [12] accuracy in identifying phrases and clauses over well-written news articles.

Our approach uses **Syntactic parsing** [16], 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 2.4 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% [17].

2.3 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.

2.3.1 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 [18–20], 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.

2.3.2 Overview

Figure 2.5 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 2.3.6). The variable binding component then searches the code to identify the variables that store the values of the sensitive input fields.

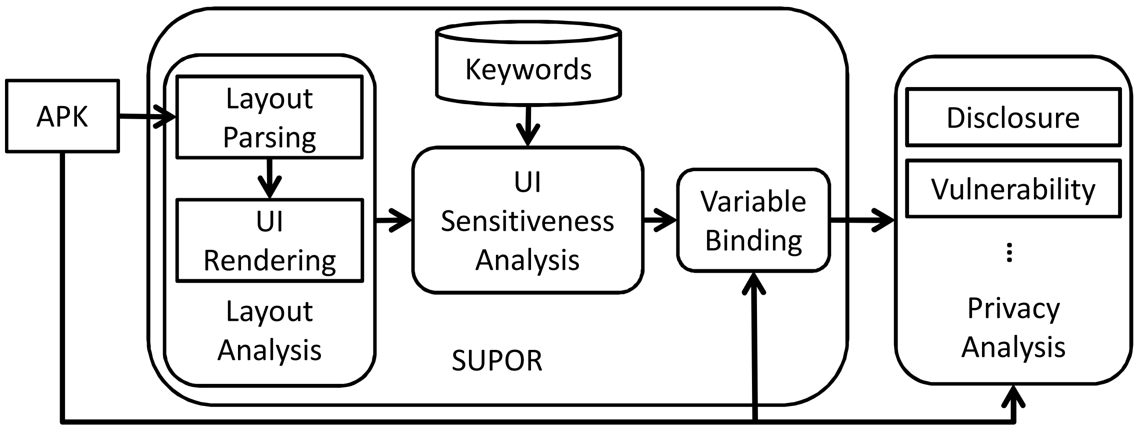


Figure 2.5.: Overview of SUPOR.

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.

2.3.3 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 2.2.3, 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 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 2.6 shows the UI model obtained by rendering the layout file in Figure 2.2. 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 2.6, 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 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,

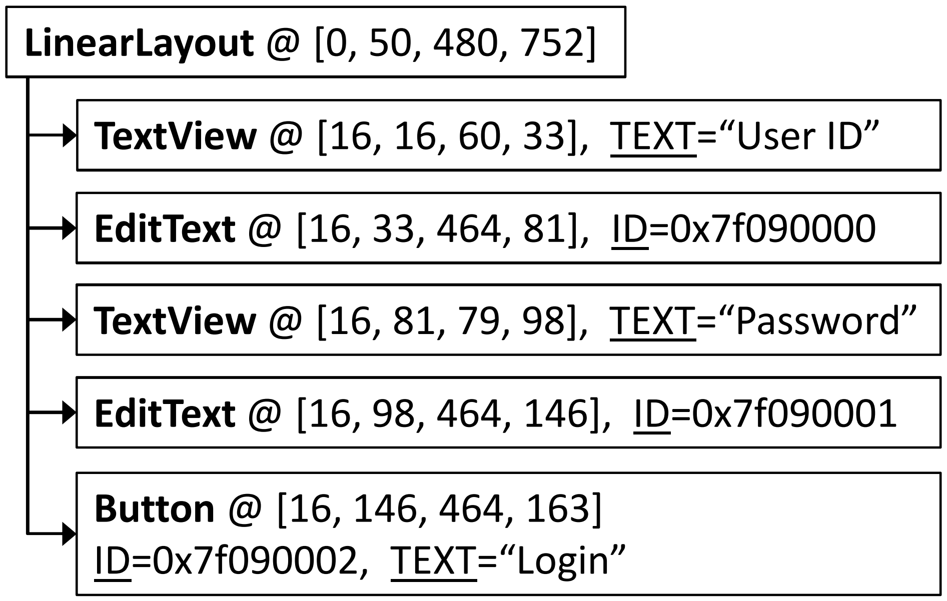

LinearLayout @ [0, 50, 480, 752]
TextView @ [16, 16, 60, 33], TEXT="User ID"
EditText @ [16, 33, 464, 81], ID=0x7f090000
TextView @ [16, 81, 79, 98], TEXT="Password"
EditText @ [16, 98, 464, 146], ID=0x7f090001
Button @ [16, 146, 464, 163]
ID=0x7f090002, TEXT="Login"

Figure 2.6.: UI model for Figure 2.1 on 480x800 screen. Only the ID, relative coordinates and text of the widgets are presented here.

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`).

2.3.4 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.

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 $KW$
```

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 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 2.7 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

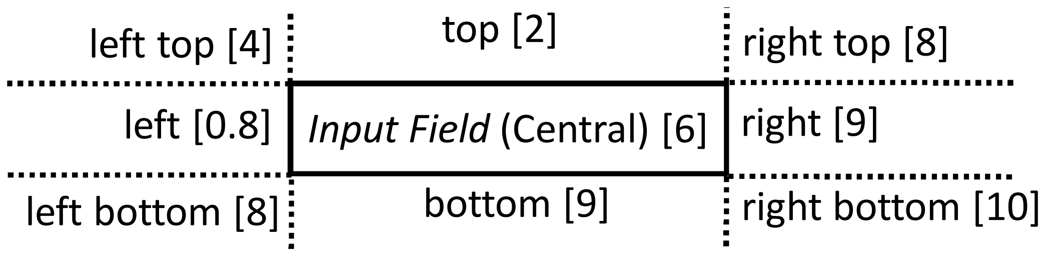


Figure 2.7.: The partition of the UI is based on the boundary of the input field.

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 2.7. 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 2.8 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

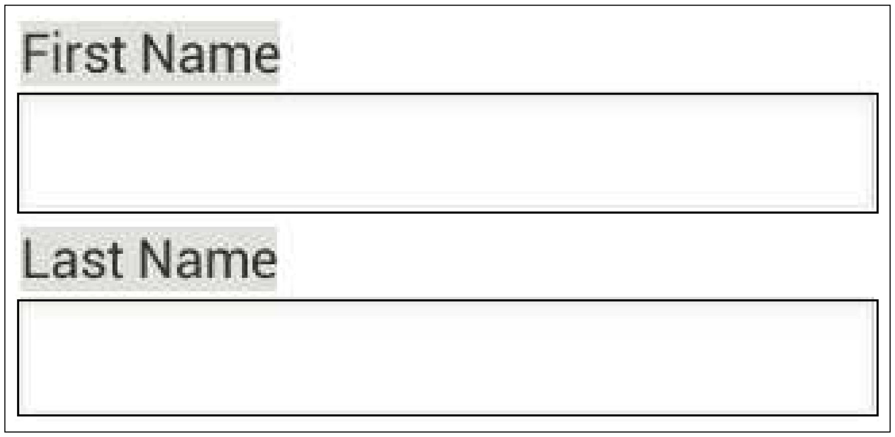


Figure 2.8.: Example for UI widget sensitiveness analysis.

Table 2.2.: Scores of the text labels in Figure 2.8.

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

shown in Table 2.2. 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`”).

2.3.5 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 2.2.2, `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 sensitive 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 2.3. 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 inter-procedural backward slice [21–24] 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 2.3, the widget variable `tx-tUid` 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 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.

2.3.6 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

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

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 [12] to parse each text line into a syntactic tree as discussed in Section 2.2.4, 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 [13, 14, 25].

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 2.3. Note that we do not use “Address” for the category “Personal Info”. Although personal address is sensitive information, our preliminary results

Table 2.3.: 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

show that this 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 2.5) show that it is a relatively complete dataset for the ten categories that we focus on in this work.

2.4 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 [26] 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 [27]. The IR is further converted to WALA [28] static single assignment format (SSA). WALA [28] 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 [3] and make several customizations to improve the effectiveness.

The 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.

To identify sensitive user inputs, SUPOR includes totally 11 source categories, including the 10 categories listed in Section 2.3.6 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).

In details, 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.

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

2.5 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.

2.5.1 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 sensitive 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 2.4 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

Table 2.4.: 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%

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.

2.5.2 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.

2.5.3 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 2.5 shows the detailed analysis results. Column “#Layouts ” counts the number of layouts containing input fields in each app, while Column “#Layouts with SIF” 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 SIF” 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 “Dup 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

Table 2.5.: UI analysis details for 20 randomly chosen apps.

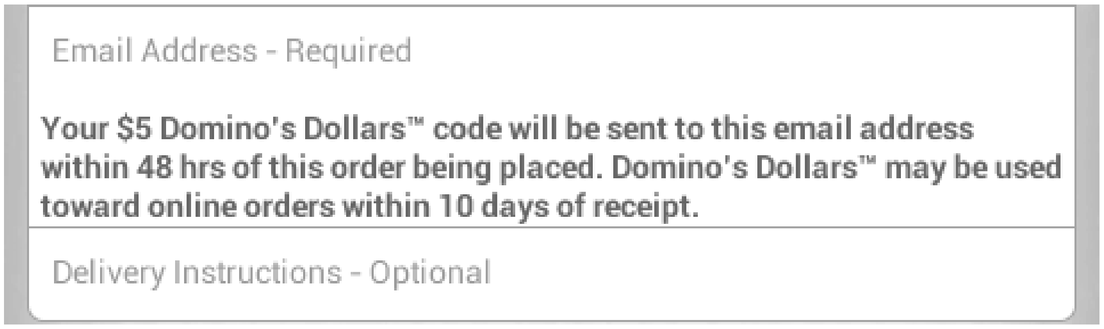
App ID	#Layouts	#Input Fields	#Layouts with SIF	#Reported SIF				FP	FN	Dup ID
				Password	Hint	Label	Total			
1	8	18	4	6	0	3	9		2	○
2	37	77	2	0	0	8	8			
3	3	3	1	0	1	0	1			
4	4	9	3	0	0	6	6			○
5	5	7	1	1	0	0	1			
6	17	52	10	6	12	12	30	1		○
7	4	5	2	0	0	3	3			
8	15	22	9	8	3	2	13	1		
9	3	7	1	1	1	0	2			
10	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	

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 [29].

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 2.9, the hint of the "Delivery Instructions" input field does not contain sensitive keywords, and thus SUPOR identifies the close text label for determining its sensitiveness. However, SUPOR incorrectly associates 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


Email Address - Required

Your $5 Domino's Dollars™ code will be sent to this email address within 48 hrs of this order being placed. Domino's Dollars™ may be used toward online orders within 10 days of receipt.

Delivery Instructions - Optional

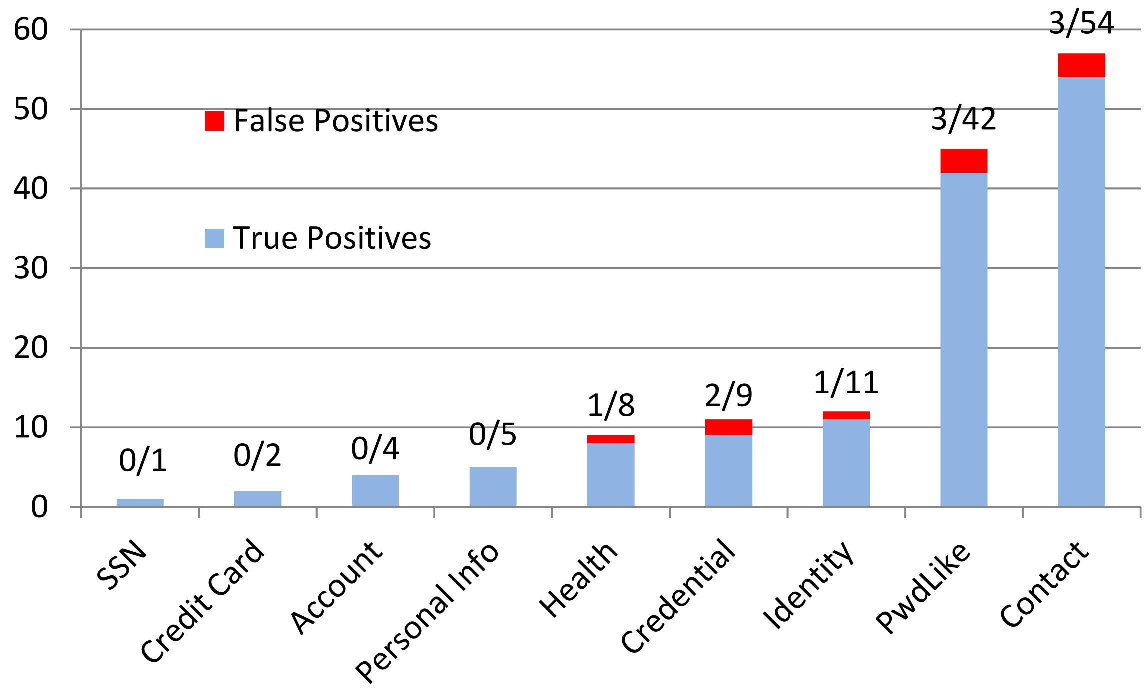
Figure 2.9.: False positive example in UI sensitiveness analysis.

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.

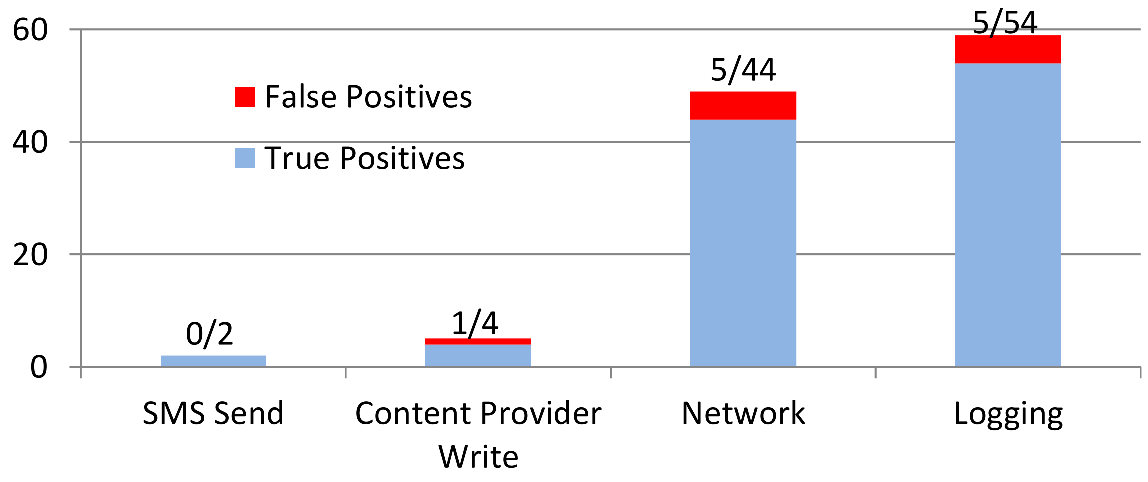
2.5.4 Accuracy of Detecting Sensitive User Input Disclosures

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 2.10 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 accuracy 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.



(a) TPs and FPs by source categories.



(b) TPs and FPs by sink categories.

Figure 2.10.: True positives and false positives by source/sink categories for the reported apps.

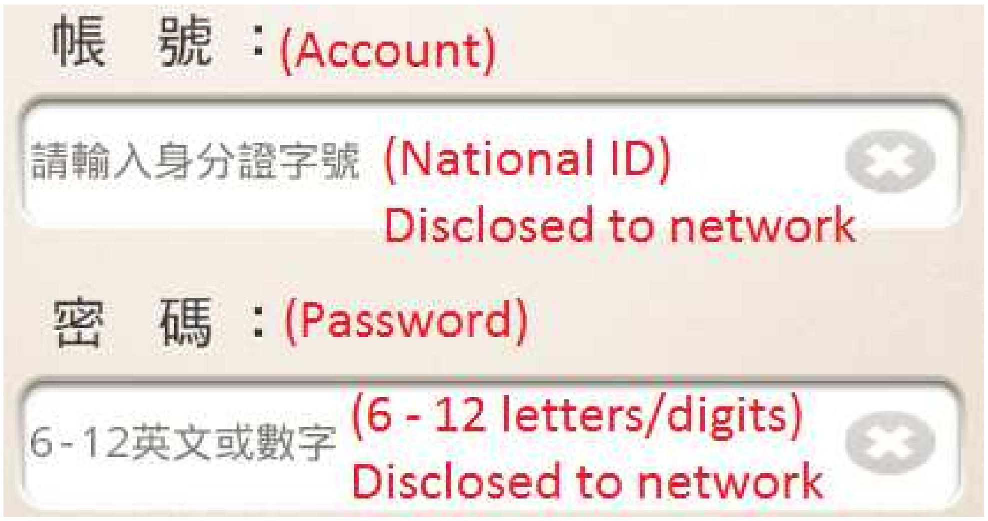


Figure 2.11.: Case study: National ID and password disclosure example without protection.

2.5.5 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 (see Figure 2.11), 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 where HTTPS connections are used 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

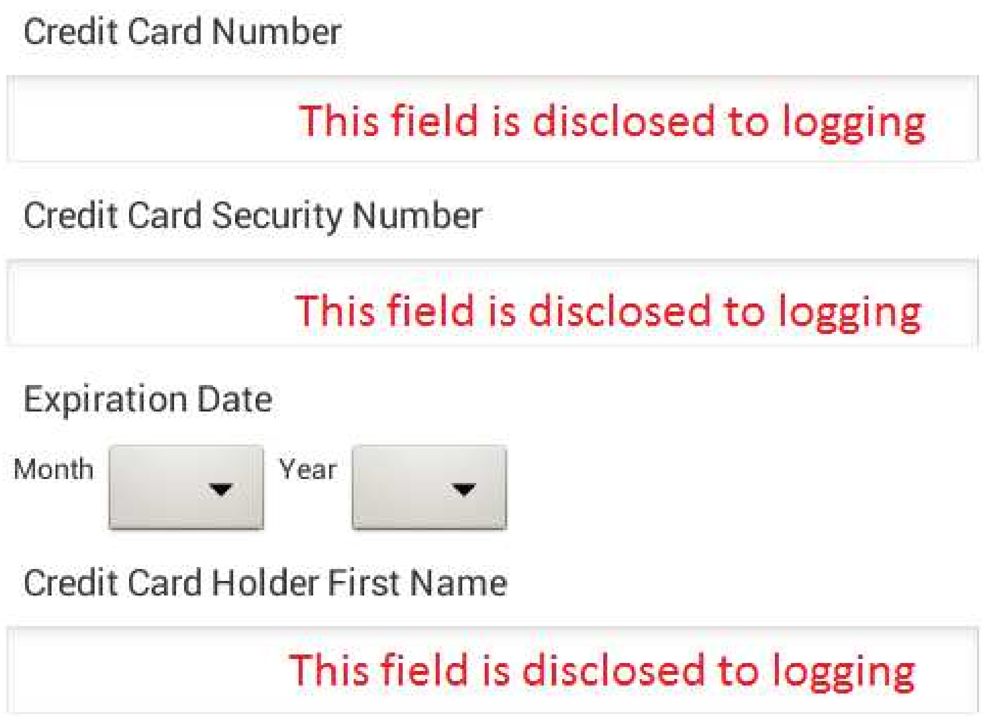

Credit Card Number
This field is disclosed to logging
Credit Card Security Number
This field is disclosed to logging
Expiration Date
Month
Year
Credit Card Holder First Name
This field is disclosed to logging

Figure 2.12.: Case study: Credit card information disclosure example.

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 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 third example app (*com.nitrogen.android*) discloses credit card information, a critical financial information provided by the users. Figure 2.12 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

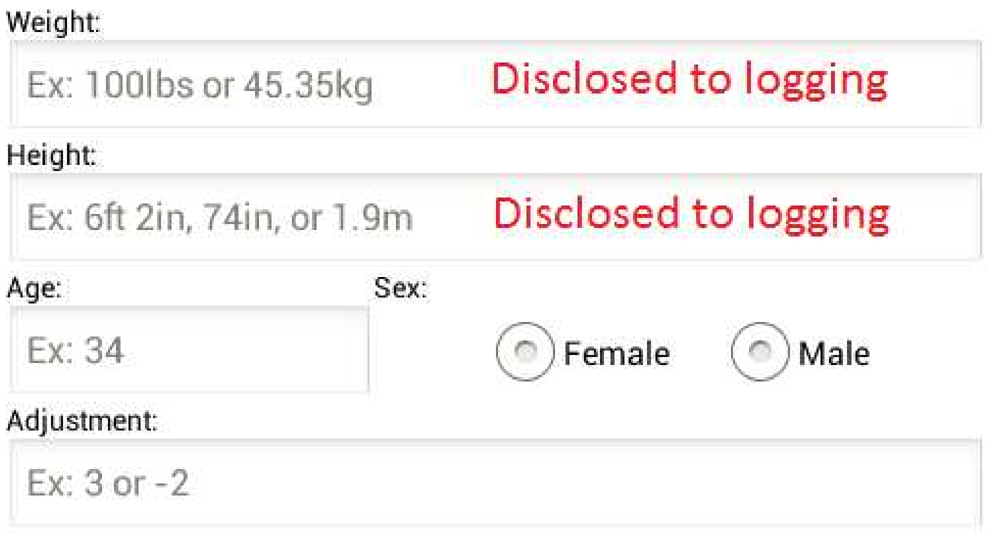


Figure 2.13.: Case study: Health information disclosure.

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.

Our last example shows that SUPOR also identifies apps that disclose personal health information to logging. Figure 2.13 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.

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 [30]). 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.

2.6 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 [29].

2.7 Related Work

Many great research works [1–4, 6, 9, 31–36] 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 [35, 36] 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 [37] 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 [38] 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 [39].

NLP techniques have been used to study app descriptions [7, 8, 15]. WHYPER [7] and AutoCog [8] leverages NLP techniques to understand whether the application descriptions reflect the permission usage. CHABADA [15] 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 [40] 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 [41]. 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 [42]. SUPOR focuses on sensitive user input identification which is different from the problems studied by these existing works.

The closest related work is UIPicker [43], 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 [10, 44–47] for GUI testing. GUITAR is a well-known framework for general GUI testing, and GUI ripper [10], 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 [44] and [46, 47], 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.

2.8 Summary

In this chapter, 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.

3 BIDTEXT: DETECTING SENSITIVE DATA DISCLOSURE VIA BI-DIRECTIONAL TEXT CORRELATION ANALYSIS

Traditional sensitive data disclosure analysis faces two challenges: to identify sensitive data that is not generated by specific API calls, and to report the potential disclosures when the disclosed data is recognized as sensitive only after the sink operations. We address these issues by developing BIDTEXT, a novel static technique to detect sensitive data disclosures. BIDTEXT formulates the problem as a type system, in which variables are typed with the text labels that they encounter (e.g., during key-value pair operations). The type system features a novel bi-directional propagation technique that propagates the variable label sets through forward and backward data-flow. A data disclosure is reported if a parameter at a sink point is typed with a sensitive text label. We have already gotten some preliminary results by evaluating BIDTEXT on some real-world Android apps. So far we have observed that the false positive rate for BIDTEXT is 10%.

3.1 Introduction

Sensitive data disclosure has been a long-standing challenge for data security. By accessing the disclosed sensitive information, adversaries can learn about the system and then conduct attack [48,49]. A prominent example is the OpenSSL Heartbleed vulnerability disclosed in 2014. The OpenSSL versions with such a flaw allow remote attackers to retrieve sensitive data, for example, user authentication credentials and secret keys [50,51]. Attackers can then compromise the target systems with the disclosed sensitive information.

The proliferation of mobile devices [52,53] makes the situation even worse since mobile devices process a lot of sensitive user data. Previous studies showed that it is common that mobile apps undesirably disclose sensitive user information [2,54–56]. Many techniques have been proposed that work at the system level or the application level, static or dynamic [1,4,31,57]. Haris *et al.* provide a comprehensive list of the approaches to detecting sensitive information disclosures in mobile computing [58]. All these approaches require definition of the sensitive data sources, usually certain APIs whose return value is sensitive. With the definition, if *forward* data flow is observed between taint sources and sinks, disclosure defects are reported. Later, researchers realized that some generic APIs may return sensitive values, depending on the context, although they may return insensitive values in many cases. SUPOR [59] and UIPicker [43] aimed to identify which user inputs on the user interfaces can be sensitive. Then the sensitive inputs are associated with the variables in the code such that static or dynamic forward data flow analysis can be applied to detect the potential sensitive user inputs disclosures. Sensitive user inputs are identified in the context of the user interfaces which contain text or graphical information to instruct what the users should enter.

However, the above solutions still have limitations. Sensitive data may come from generic API methods not related to UI (*e.g.,* loading data from some file or receiving data from network). In these cases, most existing approaches would not work properly. We cannot simply treat the generic APIs as the taint sources as that will lead to a large number of false warnings. In addition, forward data flow analysis is insufficient. In many cases, a piece of data may be first emitted through a sink and then later typed as sensitive. There may not be any forward data flow from the type revelation point to the sink point.

In this section, we develop BIDTEXT, a technique to detect data disclosures by examining the text labels correlated with variables. The text labels, either from the code (*e.g.,* the textual keys in key-value pairs) or the UI, provide rich information about the data contained in the variables. BIDTEXT extracts these labels, and leverages a novel type system to propagate these labels through both backward and forward data flow. Data disclosures are reported when a parameter at a sink point is typed with a sensitive textual label. The bi-directional propagation scheme is unique and different from the traditional unification based type inference systems. It features the capability of avoiding undesirable unification of text labels, enabling a low false positive rate. Backward propagation allows BIDTEXT to capture cases in which data sensitiveness is revealed after the data is sent through some sink.

Our work makes the following contributions:

- We propose BIDTEXT, a novel method to detect sensitive data disclosures. BIDTEXT leverages constant text labels and features a novel type system that performs bi-directional text label propagation.
- We implement a prototype of BIDTEXT for Android apps, and evaluate it on some real-world apps. So far we have observed that the false positive rate for BIDTEXT is 10%.
- BIDTEXT is available at `https://bitbucket.org/hjjandy/toydroid.bidtext`.

3.2 Motivating Example

We use a real-world Android app *com.buycott.android* to motivate our technique. It is an app that allows users to check the company/vendor of a product by scanning the product's barcode. It even allows users to view the family tree of the company/vendor. Users can then make decision on whether this is a company that rips off its customers so that they do not want to have business with. Users can also start/join campaigns against specific companies [60].

Figure 3.1 shows a piece of simplified code snippet from the app. The app sends a request to the Web server and obtains a list of post messages. The HTTP response is converted to a string in the app and then sent to a handler via a `Message` object. The following operations are present in the code snippet. At line 7, a key-value mapping is retrieved from the `Message` object. Then the data string of the message is obtained from the mapping at line 8. Right after that, the data string is written to the log file at line 9. Note that writing to a log file is usually considered as a sink for data disclosures [9, 35, 36,

```java
class CampaignActivity_20 implements Handler.Callback{
  CampaignActivity act;
  CampaignActivity_20(CampaignActivity a){
    this.act = a;
  }
  public boolean handleMessage(Message msg){
    Bundle b = msg.getData();
    String dt = b.getString("data");
    Log.d("CampaignActivity", "Got data back: " + dt);//sink
    Runnable r = new CampaignActivity_20_1(dt);
    act.runOnUiThread(r);
    return false;
  }
}
class CampaignActivity_20_1 implements Runnable{
  String jsonString;
  CampaignActivity_20_1(String data){
    jsonString = data;
  }
  public void run(){
    JSONArray jsonArray = new JSONArray(jsonString);
    int len = jsonArray.length();
    for (int i=0; i<len; i++) {
      JSONObject json = jsonArray.getJSONObject(i);
      String url = json.getString("avatar_url");
      ImageView iv = ... // omitted
      displayImage(url, iv); // omitted
      String un = "<b>" + json.getString("username") + "</b>" +
          json.getString("created_at");
      TextView tv = ... // omitted
      tv.setText(Html.formHtml(un));
      String c = json.getString("content");
      TextView ctv = ... // omitted
      ctv.setText(Html.fromHtml(c));
      // ...
    }
  }
}
```

Figure 3.1.: Motivating example from app *com.buycott.android*.

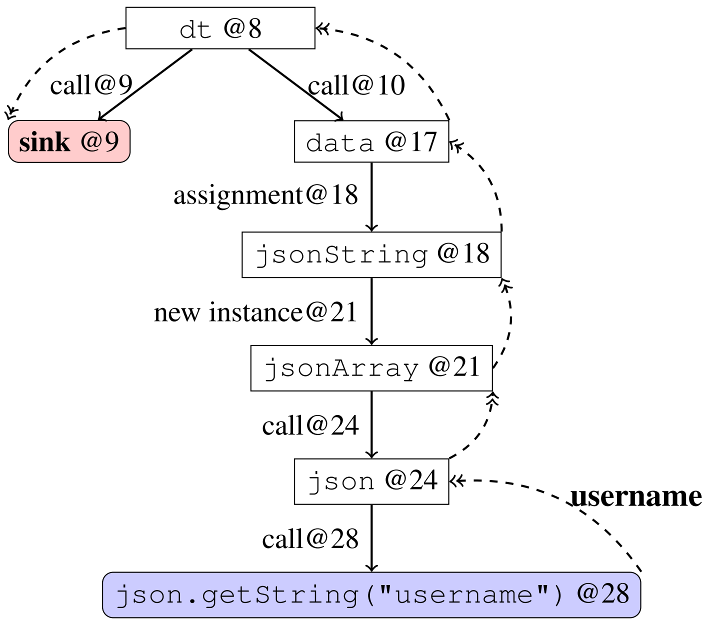


Figure 3.2.: Data flow (solid arrows) and type propagation (dashed arrows) for Figure 3.1.

59] because log files can be accessed by malware[1]. After the logging operation, the app instantiates a `Runnable` object with the data string at line 10, which runs in the UI thread (line 11) to allow interactions with UI elements.

The data string is transmitted to the `Runnable` instance via the instantiation at line 10. Inside the constructor at line 17, the data is stored in a field variable `jsonString` at line 18. When the UI thread is running, the `run()` method at line 20 is invoked. The data string is converted to a `JSONArray` object at line 21 which is then iterated. Every element in the array is a `JSONObject` (line 24). The app then obtains the URL for the avatar image, the corresponding user Id, the time of creation and the content of the post message by looking for the values via corresponding keys in the JSON object (lines 25, 28, and 31). All such information is shown on some UI elements (*e.g.,* line 33).

Now let's consider the potential sensitive data disclosure in this running example. Based on the above description, the data falling into the sink at line 9 comes from the Web server.

[1]The recent version of Android has substantially mitigated this problem by limiting access to log files. But there are still a large number of devices running old versions of Android. Note that BIDTEXT is general to support various configurations of sink points.

We later know that the data contains some sensitive user account information. In other words, the app retrieves the sensitive user account information from the server and writes it to the local log file without any encryption. This is a typical kind of undesirable information disclosure [61, 62] that emits sensitive information from server such as user account, balance in bank account, and employee salary to local files.

Traditional sensitive data disclosure analysis inspects the data flow between some sensitive source point, for example, an API call whose return value can be easily recognized as sensitive (*e.g.,*`Tele-phonyManager.getDeviceId()` in Android), and a sink point (*e.g.,* a file write or a socket send). If forward data flow can be discovered from the source point to the sink point, a disclosure problem is reported. In this example, while we do have data flow from the Web server response to the logging operation but we cannot determine whether the response contains sensitive data *from the operations along the data flow*. If we treat all data from server sensitive, a lot of false alarms will be produced; but if we simply ignore them, we miss true disclosures as in this example.

Different from the traditional disclosure analysis, our technique relies on the observation that the sensitiveness of data used in applications can be recognized through examining the textual information involved in the operations. Such texts are constant strings in either the code or the user interfaces. We randomly sampled 2,000 Android apps and found that on average each app contains 76.7 constant strings in layout files (*i.e.,* XML files used to statically define UIs) and 151 constant strings in app code. These constant strings often provide rich information about what is being held by the corresponding variables. For example, in Figure 3.1, method call `json.getString("username")` at line 28 uses a constant string "*username*". We can infer that the JSON object contains some sensitive user Id. Since the JSON object is part of the Web server response, according to the work flow, we can conclude that the response contains sensitive information. Thus the logging operation at line 9 should be reported as a sensitive data disclosure.

Note that even if we recognized that the JSON object at line 28 contains sensitive information, we could not detect the disclosure problem using traditional analysis techniques that try to find forward data flow from source points to sink points. We show the data flow via solid arrows in Figure 3.2, starting from retrieving the data from the key-value mapping (line 8). If we treat line 28 as a source point, we cannot get a forward data flow path from the source point to the sink point. Thus the disclosure defect is still missed after we augment traditional techniques with our new sensitive data recognition method.

BIDTEXT solves the problem by introducing *bi-directional* propagation. Instead of propagating tags like *tainted* and *untainted* in traditional techniques, our approach uses the constant strings as the tags and propagates both backward and forward. As the dashed arrows in Figure 3.2 show, constant text "*username*" is propagated backward from the method call at line 28 to the variable `json` created at line 24, and so on. Consequently, variables `jsonArray`, `jsonString`, `data` and finally `dt` are tagged with the text "*username*". Intuitively, it means all these variables contain sensitive user Id information. Next we forwardly propagate the tag from line 8 to the sink point at line 9. Therefore, the logging statement operates on variables that are associated with text "*username*". By applying this approach to the whole code snippet, we obtain the set of correlated text as {"*Campaign-Activity*", "*Got data back:*", "*data*", "*avatar_url*", "*username*", "*created_at*", "*content*"}. The first two textual tags are associated to the variable directly at the sink point. Tag "*data*" is propagated to the variable (at the logging statement) in a forward manner. The remaining texts are propagated to the sink point via a bi-directional manner discussed above.

BIDTEXT also associates UI texts to variables. UI often contains texts that also indicate the sensitiveness of data shown on the UI (see [43, 59]). We examine the corresponding layout file to get the texts, add them to the tag set of the related variables and propagate them like the texts found in the code. In the example, we can find several code locations that interact with the UI (*e.g.,* line 33), through which we identify the corresponding layout files to collect UI texts. However, the content of the UI is dynamically created and none of the UI elements holds constant texts. Therefore, no GUI texts are propagated to the sink point in this example.

Next we apply a natural language processing (NLP) technique to the tag set of the sink point to find out if the texts can tell the sensitiveness of the variable `dt`. Among the

Program	p ::=	s*	
Statement	s ::=	v := t	/*constant string in code*/
	|	v := i	/*UI-related Id*/
	|	v := c	/*values of other types*/
	|	$v := \ominus v_1$	/*unary assignment*/
	|	$v := v_1 \oplus v_2$	/*binary assignment*/
	|	$call(m,v_a \rightarrow v_f)$	/*v_a/v_f actual/formal arg*/
	|	$v := return(m,v_r)$	/*m returns v_r to v*/
	|	$v := apicall(m,v_a)$	/*API call to method `m`*/
	|	$IF(v)\ \{s_t\}\ ELSE\ \{s_f\}$	
	|	LOOP {s}	/*loop structure*/
	|	$v := \phi(v_t, v_f)$	/*value merging in SSA*/
Variable	v		
Method	m		
String	t		
ID	i		
Value	c		/*Non-str, non-Id Values*/

Figure 3.3.: Language.

collected texts, "*username*" matches a predefined sensitive keyword. Thus our technique reports a sensitive data disclosure problem for the logging operation at line 9.

3.3 Design

We propose BIDTEXT, a static bi-directional text correlation analysis approach, to detect sensitive data disclosures. BIDTEXT combines both the bi-directional propagation and the new approach that uses internal constant texts to identify sensitive variables as illustrated in Section 3.2.

3.3.1 Language Abstraction

To simplify our discussion, we introduce an abstract language. The language is presented in Figure 3.3. We only model the language features that are related to explaining the text correlation analysis and the bi-directional propagation. Others are abstracted away or simplified. As we discussed in Section 3.2, we leverage the constant texts in the code as well as in the UI to tag variables and determine whether sensitive data is disclosed at sink points. Therefore, constant strings in the code and constant Ids that are associated with UI are of special interest and explicitly modeled in the language. For simplicity, we do not allow constant strings/Ids to appear in complex operations, *e.g.,* binary operations and method calls. For such scenarios, the constant is first assigned to a variable, which is further used in the complex operation. This is similar to how Android apps handle constant values in DEX bytecode. For example, the method call `json.getString("username")` at line 28 in Figure 3.1 is converted to two statements:

```
tmp = "username";
json.getString(tmp);
```

An invocation to method m(v_f) is modeled by two separate statements: call(m, $v_a \to v_f$) passing the actual argument v_a to the formal argument v_f and v=return(m, v_r) returning the value in v_r in `m()` to v in the caller. The separation allows us clearly model the data flow at the entry and the exit of a method call. v := apicall(m, v_a) abstracts invocation to an API function `m()` whose implementation is usually excluded or not available during analysis, *e.g.,* the runtime C library and the framework methods for Android apps.

The language also supports conditional branches and loops. There are different loop structures such as `for` loops and `while` loops. We ignore these differences and use a `LOOP` statement to model them. Loop conditions are not relevant to our analysis and hence not modeled. Any side effects (in the loop conditions) are explicitly modeled as assignments in the loop body.

Our language is a kind of SSA language so that ϕ function is used to merge values from different branches (of a predicate). As we will show later in Section 3.3.2, ϕ functions require delicate consideration during bi-directional propagation.

3.3.2 Type System and Bi-directional Propagation

As discussed earlier, we use the constant texts in either the code or the UI to tag the correlated variables and propagate the tags bi-directionally. We formalize this approach in a type system, *i.e.,the set of tags associated with a variable is treated as the type of the variable*. Since the type is a set, we also call it a *type set* in this paper. The mappings from variables to their type sets form the context Γ of the type system, which is iteratively updated during analysis until a fixed point is reached. For example, at the beginning, Γ is empty. Upon a statement `tmp = "username"`, Γ is updated to $\{tmp : \{username\}\}$. At this point, we have $\Gamma \vdash tmp : \{username\}$, which means under context Γ, variable `tmp` is typed with set $\{username\}$. In other words, $\Gamma(tmp) = \{username\}$, where $\Gamma(tmp)$ evaluates variable `tmp` in the context to obtain the corresponding type set.

When a statement is evaluated, the context may be updated. We use $\Gamma, S \models \Gamma \Rightarrow \Gamma'$ to indicate that under context Γ, evaluating statement S updates the context from Γ to Γ'.

We use $[var : T]\Gamma$ to represent an update to the context. Specifically, if no mapping is found for variable `var` in context Γ, the mapping is added into the context. But if there exists some mapping for `var`, the rule substitutes the existing type set for `var` with the given type set T. Multiple mappings can be updated simultaneously, *e.g.,* $[var : T, var' : T']\Gamma$ updates the context for two variables `var` and `var'`.

Given two type sets T and T', $T \cup T'$ unions the two sets while $T - T'$ returns a new type set which contains all elements belonging to T but not T'.

With the language in Figure 3.3 and the above definitions, we define the bi-directional type set propagation rules in Figure 3.4. The propagation is iterative. That means once the analysis starts, it does not terminate until the context Γ reaches a fixed point.

$$\text{Const-Binding}\ \frac{}{\Gamma, v := t \models \Gamma \Rightarrow [v : \{t\}]\Gamma}$$

$$\text{UI-Binding}\ \frac{resource\_id(i)}{\Gamma, v := i \models \Gamma \Rightarrow [v : extract\_text(i)]\Gamma}$$

$$\text{Unary-Assignment}\ \frac{\Gamma \vdash v : T \qquad \Gamma \vdash v_1 : T'}{\Gamma, v := \ominus v_1 \models \Gamma \Rightarrow [v : T \cup T', v_1 : T' \cup T]\Gamma}$$

$$\text{Binary-Assignment}\ \frac{\Gamma \vdash v : T \qquad \Gamma \vdash v_1 : T_1 \qquad \Gamma \vdash v_2 : T_2}{\Gamma, v := v_1 \oplus v_2 \models \Gamma \Rightarrow \left[\begin{array}{l} v : T \cup T_1 \cup T_2, \\ v_1 : T_1 \cup (T - T_2), \\ v_2 : T_2 \cup (T - T_1) \end{array}\right]\Gamma}$$

$$\text{Phi-Assignment}\ \frac{\Gamma \vdash v : T \qquad \Gamma \vdash v_1 : T_1 \qquad \Gamma \vdash v_2 : T_2}{\Gamma, v := \phi(v_1, v_2) \models \Gamma \Rightarrow \left[\begin{array}{l} v : T \cup T_1 \cup T_2, \\ v_1 : T_1 \cup (T - T_2), \\ v_2 : T_2 \cup (T - T_1) \end{array}\right]\Gamma}$$

$$\text{Method-Call-Param}\ \frac{\Gamma \vdash v_a : T \qquad \Gamma \vdash v_f : T'}{\Gamma, call(m, v_a \rightarrow v_f) \models \Gamma \Rightarrow [v_f : T' \cup T, v_a : T \cup T']\Gamma}$$

$$\text{Method-Call-Return}\ \frac{\Gamma \vdash v : T \qquad \Gamma \vdash v_r : T'}{\Gamma, v := return(m, v_r) \models \Gamma \Rightarrow [v : T \cup T', v_r : T' \cup T]\Gamma}$$

$$\text{API-Call}\ \frac{\Gamma \vdash v_a : T' \qquad \Gamma \vdash v : T}{\Gamma, v := apicall(m, v_a) \models \Gamma \Rightarrow \left[\begin{array}{l} v : T \cup model\_fwd(m, v_a), \\ v_a : T' \cup model\_bwd(m, v) \end{array}\right]\Gamma}$$

Figure 3.4.: Bi-directional propagation rules.

Binding Constant Value

As mentioned earlier, we focus on constant texts in the code and the constant Ids that are associated to UI. An assignment of a constant string to a variable adds a new mapping from the variable to a set holding the string to the context. For a constant Id, we need to make sure the Id is indeed a resource Id (*e.g.,* layout Id in Android apps or an Id for a specific UI element). This check is modeled by predicate `resource_id()`. If the prerequisite satisfies, updating the context is similar to the constant string assignment, except that the type set is the extracted texts from the corresponding UI through function `extract_text()`. For instance, if the constant Id is associated with a typical login screen, the extracted text set may often be $\{$*Username*, *Password*, *Login*$\}$.

Propagation for Assignment

Rule Unary-Assignment updates the context for both the LHS and RHS variables with the union of the two separate type sets. Note that it allows the tags from LHS to propagate to RHS and vice versa through the union operation (*i.e.,* bi-directional propagation). Use the statement `jsonString = data` at line 18 in Figure 3.1 as an example. Assume before evaluating this statement, $\Gamma(jsonString) = \{avatar\_url, username, created\_at, content\}$ and $\Gamma(data) = \{data\}$ via previous evaluation steps. After evaluating this statement, the type sets for both variables `jsonString` and `data` are updated to $\{avatar\_url, username, created\_at, content, data\}$. This shares some similarity with type unification in classic type inference. However, as we will see next, unification does not properly model the intended propagation behavior for binary operations and ϕ functions.

For a binary assignment, we cannot simply union all the type sets of the LHS and RHS variables and associate the resultant type set to all the variables, which is what classic type inference would do. We observe that this is undesirable as it allows the type set of a RHS variable to be propagated to another RHS variable while the operation does not induce any data flow between the two variables. Intuitively, assuming the two RHS variables are v_1 and v_2, v_1 being associated with a sensitive tag does not entail v_2 having the same sensitive tag (by the operation). Thus, as specified by Rule Binary-Assignment, the propagation is conducted as follows. The type sets of the RHS variables are unioned and inserted to the type set of the LHS variable. Only the part of the LHS type set that is not in the type set of v_1 is propagated to v_2 and only the part of the LHS type set that is not in the type set of v_2 is propagated to v_1. There is a corner case in which the two RHS variables are the same one, *e.g.,* `a = b ⊕ b`. The updated type set for `b` is $\Gamma(b) \cup (\Gamma(a) - \Gamma(b))$, which is equal to $\Gamma(a) \cup \Gamma(b)$. In other words, this special case behaves the same as a unary assignment. The propagation for ϕ statements has the same nature (Rule Phi-Assignment).

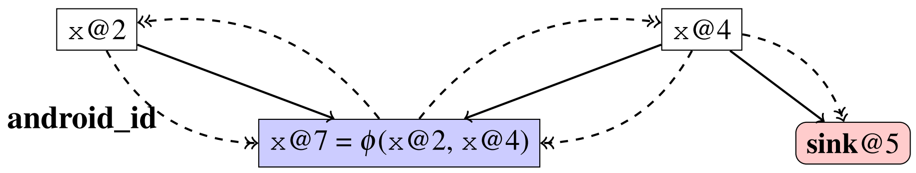
We use a real example from an Android app *com.mojo.animewallpaper* to show how our propagation rule for ϕ statements eliminates false alarms. The simplified code snippet is shown in Figure 3.5a. If a certain condition satisfies, the device Id is assigned to variable `x` at line 2. The detail of acquiring the device Id is omitted but eventually a constant string "*android_id*" is added to the type set of `x`. If the condition doesn't satisfy, a random value is generated as the requested Id at line 4 and stored to variable `x`, which is immediately used at a sink point at line 5. After the branch, variable `x`, whose value is either the real device Id or a random value, is used elsewhere.

From the perspective of ϕ representation, we know that right before the `x` is used at line 7, we have a ϕ statement as `x@7` = ϕ(`x@2`, `x@4`). The data flow for the several occurrences of `x` is described by the solid arrows in Figure 3.5b and the propagation relations are shown by dashed arrows.

Consider a naive bi-directional propagation that simply unions all the type sets. During the first iteration, "*android_id*" is propagated to `x@7` via forward propagation. Nothing is backwardly propagated to `x@2` or `x@4` from `x@7`. Therefore, at the end of the first iteration, $\Gamma(x@2) = \Gamma(x@7) = \{android\_id\}$ and $\Gamma(x@4) = \emptyset$. Then during the second iteration, if we directly propagate the type set of `x@7` to both `x@2` and `x@4`, we would get $\Gamma(x@4) = \{android\_id\}$, which is later propagated to the sink point at line 5. Thus a sensitive data disclosure is reported which is a false alarm. In contrast, our propagation rule supports the mutual exclusion of the type sets in the two respective branches. Specifically, we only backwardly propagate $\Gamma(x@7) - \Gamma(x@2)$, *i.e.,* an empty set, to `x@4`. At last, the

```
if (...) {
  x = getDeviceId(); // x is tagged with "android_id"
} else {
  x = some_random_uuid(); // gen random value for x
  Log.d("Random: ", x); // sink
}
use(x);
```

(a) Simplified code snippet.



(b) Data flow and type propagation.

Figure 3.5.: App *com.mojo.animewallpaper*: code example and bi-directional propagation for ϕ.

type set of x@4 stays unchanged and the sink point does not observe any sensitive type for the variable. Thus no sensitive data disclosure is reported.

Propagation for Method Calls

Propagation through a method call occurs at passing argument from the caller and returning value from the callee. Therefore, we define two separate rules for these two events. Note that these two rules handle method calls whose implementations are included in the analysis. We also propose a special rule for propagation over API functions, the implementations of which are typically invisible during analysis.

Rules Method-Call-Param and Method-Call-Return union the type sets. A concrete example for rule Method-Call-Param is the instantiation call at line 10 in Figure 3.1. The constructor at line 17 is invoked and the value held by variable dt is passed to variable `data`. Then constant value "*data*" associated with `dt` is propagated to `data` and "*username*" associated with `data` is backwardly propagated to `dt`.

Rule API-Call does not directly propagate the type sets between parameters and the return value. BIDTEXT relies on the model for the API function for proper propagation. Prior static taint analysis [4,35,36] have shown that it is effective to simply propagate from all parameters to the return value and the receiver object (*i.e.,*`this` reference in instance method calls). However, this naive approach does not work well in bi-directional propagation. We need to investigate the type correlations for the variables involved in an API call, including all the parameters and the return value.

Some API functions may not support fully bi-directional propagation among the variables. For example, variable `name` can be used to type `value` in statement `value=HashMap.get(name)` but not the reverse according to the semantics. Specifically, if `name` holds some sensitive constant strings, we can infer that `value` may hold sensitive information, but not the other way around. If we ignore `this` reference, after evaluating the statement under context Γ, we have $\Gamma'(name) = \Gamma(name)$ and $\Gamma'(value) = \Gamma(value) \cup \Gamma(name)$. Many API functions, on the other hand, can be applied with the naive propagation policy, unioning the type sets of all variables. For example, we have $\Gamma'(ret) = \Gamma'(str) = \Gamma(ret) \cup \Gamma(str)$ after evaluating statement `ret = str.toUpperCase()` under context Γ. In the rule, the behavior depends on functions `model_fwd()` and `model_bwd()` which define the propagation policies from v_a to v and from v to v_a, respectively.

We formalized our approach to identifying and bi-directionally propagating constant texts in a type system and developed a set of propagation rules based on our abstract language in Figure 3.3. While the rules are general for our language, in practice we need to perform a number of enhancements to the rules to handle real-world language/program features. These enhancements are discussed in next section.

$$\text{CheckAlert} \quad ::= \quad \text{IF}(\text{v}_c)\ \{\text{alert}(\text{v}_m)\}$$

(a) Specialized statement.

$$\text{Check-Alert}\ \frac{\Gamma \vdash v_c : T \qquad \Gamma \vdash v_m : T'}{\Gamma, IF(v_c)\{alert(v_m)\} \models \Gamma \Rightarrow [v_c : T \cup T']\Gamma}$$

(b) Propagation rule.

Figure 3.6.: Abstraction and propagation rules for check-and-alert cases.

3.3.3 Practical Enhancements

There are two main practical enhancements to our formal model that are critical to the effectiveness of BIDTEXT.

Check and Alert

It is common in real programs to prompt some alerts to the user or write to the log file if a condition check fails. In this case, we can use the alert/log message to infer what the corresponding variables involved in the condition check may hold. For example, an Android app can alert the user about some previous errors, *e.g.*, some required inputs are missing, by showing a short message on the screen. A typical implementation looks like the following.

```
if (str == null || str.isEmpty()) {
   Toast.makeText(this, "Please Enter Password", 1);
}
```

We can type variable `str` with the constant text "*Please Enter Password*" and propagate it through the aforementioned rules.

The abstraction and the corresponding propagation rule are shown in Figure 3.6. This applies to a set of API functions, called the *alert* functions.

String Concatenation

String concatenation is common in real-world apps. A concatenation operation may involve both constant values and multiple variables. If we simply union the type sets of all the involved variables and update the variables with resultant type set, we may introduce false positives. Furthermore, the associations between the constant strings (involved in the concatenation) and the variables (involved in the concatenation) also need to be properly identified. A simple strategy that associates all constant strings to all variables also produces a lot of false positives. For example, building a URL often involves multiple variables, each holding a value as part of the HTTP request. The variables can be either sensitive (*e.g.,* password) or insensitive (*e.g.,* user comment). We need to distinguish the exact types correlated to the variables. Consider the following example, in which a typical URL is constructed.

```
url = "http://.../login?username=" + un + "&pwd=" + p;
```

At the bytecode level, the above statement is converted to:

```
builder0 = new StrinBuilder("http://.../login?username=");
builder1 = builder0.append(un);
builder2 = builder1.append("&pwd=");
builder3 = builder2.append(p);
url = builder3.toString();
```

Assume the model for API `StringBuilder.append()` entails fully bi-directional propagation, *i.e.,* we propagate the type sets of all involved variables to each other. The constant string "*http://.../login?username=*" is propagated to `builder0`, `builder1`, `un`, `builder2` and `p`. A later text analysis would indicate that both `un` and `p` are associated with the sensitive text "*username*", which is incorrect for variable `p`. Similarly, "*&pwd=*" will be propagated to `un`, which causes a false alarm.

However, if we do not allow the propagation from the `StringBuilder` instance (*e.g.,* `builder0`) to the appended variable (*e.g.,*`un`), that is, the red and blue edges are

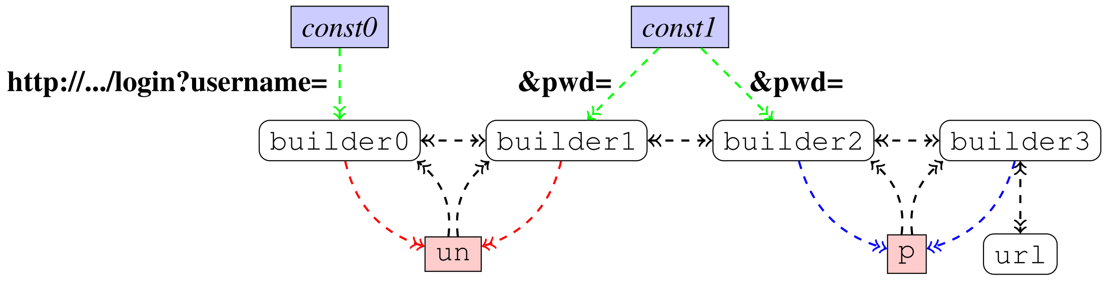


Figure 3.7.: Propagation graph for a simple string concatenation.

$$\text{Str-API}\ \frac{api\_w\_str(m) \qquad \Sigma \vdash v_a : E \qquad \Gamma' = string\_partition(m, E)}{\Gamma, v := apicall(m, v_a) \models \Gamma \Rightarrow \Gamma \cup \Gamma'}$$

Figure 3.8.: Propagation rules for string concatenation.

removed from Figure3.7, then neither "*username*" nor "*pwd*" could be propagated to `un` or `p`. As a result, we cannot infer that these two variables may hold sensitive information.

The expected propagation, according to the semantics of the URL string, is that "*username*" is propagated to `un`, and "`pwd`" to `p`, exclusively. We observe that it is impossible to enforce such propagation through API models (*e.g.,* the model for `append()`) as an API call may only represent a local operation that does not have the global view of the concatenated string. To address the problem, we need to analyze the entire concatenated string produced at the end. In our example, we ought to examine the final result associated with `url` in order to associate the appropriate text to variables `un` and `p`. Therefore, we need to enhance our type system with the following string analysis.

Rule Str-API in Figure 3.8 determines if an API call has a string argument v_a with a well-defined format through function `api_w_str()`. For example, `new URL(str)` is such a function as it implies the variable `str` is a string of the url format. If so, the string is of interest. BIDTEXT computes an *abstract string E* for v_a, which is stored in a

$$\text{Strcat}\ \frac{\Sigma \vdash v_1 : E_1 \qquad \Sigma \vdash v_2 : E_2}{\Sigma, v := strcat(v_1, v_2) \models \Sigma \Rightarrow [v : E_1 \cdot E_2]\Sigma}$$

$$\text{Strcat-Nil}\ \frac{\Sigma \vdash v_1 : nil \qquad \Sigma \vdash v_2 : E_2}{\Sigma, v := strcat(v_1, v_2) \models \Sigma \Rightarrow [v : v_1 \cdot E_2]\Sigma}$$

$$\text{Str-Const-Assign}\ \frac{}{\Sigma, v := t \models \Sigma \Rightarrow [v : t]\Sigma}$$

$$\text{Str-If}\ \frac{\begin{array}{cc}\Sigma, s_t \models \Sigma \Rightarrow \Sigma_t & \Sigma, s_f \models \Sigma \Rightarrow \Sigma_f \\ \Sigma_t \vdash v : E_t & \Sigma_f \vdash v : E_f\end{array}}{\Sigma, IF(*)\{s_t\}ELSE\{s_f\} \models \Sigma \Rightarrow [v : E_t \mid E_f]\Sigma}$$

$$\text{Str-LOOP-Closure}\ \frac{\bot, s \models \bot \Rightarrow \Sigma' \qquad \Sigma' \vdash v : v \cdot E \qquad \Sigma \vdash v : E_0}{\Sigma, LOOP\{s\} \models \Sigma \Rightarrow [v : E_0 \cdot (E)^*]\Sigma}$$

$$\text{Str-LOOP-Simple}\ \frac{\bot, s \models \bot \Rightarrow \Sigma' \qquad \Sigma' \vdash v : E \qquad v \notin E}{\Sigma, LOOP\{s\} \models \Sigma \Rightarrow [v : E]\Sigma}$$

Figure 3.9.: Computing abstract strings.

string context Σ that maps a variable to an abstract string. An abstract string is *a regular expression including both constant strings and variables*. The abstract string is partitioned by the function `string_partition()` so that the variables in the regular expression are associated with the appropriate texts. For the above example, the rule produces Γ' = $\{un : \{username\}, p : \{pwd\}\}$. We then combine Γ' into the current context Γ and further propagate the generated texts. Next, we will first explain how the abstract strings are computed and then the `string_partition()` function.

The rules for computing abstract strings are shown in Figure 3.9. The interpretation of the rules is similar to that for our type system. One difference is that we use the string context Σ instead of the type context Γ. Rule Strcat simply concatenates the two abstract strings of the operands. Rule Strcat-Nil handles the case in which the first operand does not have any mapping, meaning that it is a string variable encountered for the first time. In this case, the variable itself is concatenated to the resulting string. It is similarly handled when the second operand does not have mapping and the rule is elided. Rule Str-Const-Assign handles the constant string assignment.

Rule Str-If specifies that for a conditional statement, BIDTEXT computes the string contexts for the true and false branches separately. For any variable that is present in the string context(s), the resulting abstract string is an alternation of the abstract strings in the branches. Consider the following code snippet.

```
if(c) {
   str := strcat("&UserId=", uId);
} else {
   str := strcat("&sessionId=", sId);
}
```

The abstract string for variable `str` is (“*&UserId=*”·$\underline{\texttt{uId}}$) | (“*&ses-sionId=*”·$\underline{\texttt{sId}}$).

Rule Str-LOOP-Closure specifies that for a loop, BIDTEXT first computes the string context for the loop body with an empty string context and then aggregates the resulting abstract strings to the original string context. In particular, if the abstract string for a variable v also contains v, it indicates the resulting string has recursive structure (caused by the loop), BIDTEXT hence associates v to a kleene closure in the context outside the loop. Tail recursion is similarly handled. Currently, BIDTEXT only handles regular languages, which is sufficient for most cases we encountered. Rule Str-LOOP-Simple specifies that if there is no recursive structure, the abstract strings are simply copied from the context of the loop body to the context outside the loop. For the following example, BIDTEXT produces the abstract string “*Output:*”·(“*A*”)* for variable `str`.

```
str := "Output:";
for (...) {
   str := strcat(str, "A");
}
```

As shown by Rule Str-API, the abstract string at an API that specifies the format of the string is partitioned to acquire the texts for the variables within the abstract string. This is done by calling `string_partition`(). This function has a number of built-in parsers that can parse the different string formats based on the API name. For example, if the API is `URL()`, it uses the parser for url. Particularly, the parser searches for symbol "*?*", the part after the symbol is parsed by "*([^=]*)=([^&]*)*" with the first part being the key and the second part the value. If the key is a constant t and the value is a variable v, Γ is updated with the mapping from v to t. BIDTEXT also has parsers for other formats such as SQL queries. For example, two mappings $\{v1 : \{password\}, v2 : \{userid\}\}$ can be extracted from an abstract string denoting a SQL update "*update TABLE set password=*"$\cdot$ $\underline{\texttt{v1}}\cdot$ " *where userid=*"$\cdot$ $\underline{\texttt{v2}}$.

For the prior URL example, `append()` is essentially a `strcat()`. According to the rules, the final abstract string for `url` is "*http://.../login?username=*"$\cdot$ $\underline{\texttt{un}}\cdot$ "*&pwd=*"$\cdot$ $\underline{\texttt{p}}$. It is partitioned so that `un` is mapped to $\{username\}$ and `p` is mapped to $\{pwd\}$.

3.3.4 Disclosure Analysis

After the type set computation converges, BIDTEXT checks whe-ther arguments at the sinks points hold any sensitive data via textual analysis. If the type set information indicates the sensitiveness of an argument, we report a potential disclosure.

The process to determine the sensitiveness of a variable with a set of associated constant texts is presented in Algorithm 2, which assumes the text set T and a set of sensitive keywords *KWD*. For each collected string (*i.e.,* word, phrase or sentence), BIDTEXT first conducts some preprocessing. For example, "*EmailAddress*" is converted to "*email address*". If a string contains more than one sentence, it is split using the standard sentence division method implemented in Stanford Parser [12]. If the string matches any keyword, we check whether it is a single word. If so, we put the string into S which holds all sensitive strings. S can be used to decide what sensitive information is disclosed after the algorithm finishes. If the string is a phrase or a sentence, we need to check if it is the negation of a

Algorithm 2 Sensitiveness determination.

```
determine_sensitiveness($T$, $S$, $KWD$)
1: for all $t \in T$ do
2:     $t'$ = preprocess($t$)
3:     if $t'$ matches in $KWD$ then
4:         if $t'$ is a word or $t'$ doesn't match any negation template then
5:             $S = S \cup t$
6:         end if
7:     end if
8: end for
```

sensitive keyword. For example, "*do not enter password here*" tells the user that the input field should not contain any password. Even though the string matches a sensitive keyword "*password*", we do not consider it sensitive. So if the corresponding variable does not have any other associated sensitive texts, it is treated insensitive and the sink does not have a sensitive data disclosure problem.

We use Stanford Parser [12] to parse a phrase or a sentence into a syntax tree, which is then converted to a dependency relation (please refer to [63]). Based on the dependency relation, BIDTEXT searches the negation word "*not*" and then checks the auxiliary word right before the negation word. It also examines if there exists a subject noun word before the auxiliary word. By combining the auxiliary word and the possible subject word, BIDTEXT can identify whether the phrase/sentence is imperative or declarative. For example, "*do not*" and "*you should not*" are imperative negations but "*you did not*" is declarative negation. BIDTEXT only considers the imperative negation as a negation (of sensitive keyword). In such cases, the text is not sensitive.

3.4 Implementation

We implemented BIDTEXT to detect sensitive data disclosures in Android apps. BIDTEXT is built on top of WALA [28], which parses the Android DEX bytecode to intermediate representations. We implemented the algorithm in [3] to collect possible entry points (*e.g.,*`onCreate` for an *activity*) in the target Android app. For each entry point, BIDTEXT builds the call graph and the dependency graph. The constant strings are propagated on the graphs. We do not distinguish the correlated text for each UI element as in [59]. Instead, all elements in one layout file are associated with all the texts found in that layout file.

BIDTEXT relies on a keyword set to determine the sensitiveness of computed texts. To acquire the keyword set, we ran BIDTEXT on 2,000 randomly selected apps and extracted all texts discovered for each sink. We then manually inspected these texts to construct the keyword set. In order to detect traditional data closures that are due to dataflow between source APIs and sink APIs instead of texts, we assign some sensitive textual keywords to the source APIs that must expose sensitive information so that BIDTEXT can propagate the keywords. For example, we assign "*imei*" to API `TelephonyManager.getDeviceId()`.

We leverage Stanford Parser [12] as the engine for analyzing phrases and sentences. BIDTEXT currently only supports English.

For better efficiency, BIDTEXT also performs on the fly type set reduction. Specifically, when a text set reaches a certain size, garbage collection is conducted by filtering out the texts in the type set that do not indicate sensitiveness and those that are redundant.

3.5 Evaluation

All experiments are performed on an Intel Core i7 3.4GHz machine with Ubuntu 12.04. The task of analyzing each app is given the maximum memory of 10GB and the maximum analysis time of 20 minutes. The subjects are a collection of 10,000 Android apps downloaded from Google Play in March 2015. The sink points used in the evaluation contain all the logging operations in Android and the Apache HTTP access APIs that are commonly used in Android apps. This is also the standard setup for many existing static taint analysis [40, 59]. The other types of sink points can be easily added to BIDTEXT.

3.5.1 Pilot Study

As discussed earlier, BIDTEXT heavily relies on accurate propagation models for API method calls. However, Android framework contains thousands of API functions, making it almost infeasible to manually build the models for all API functions. Our approach is to randomly select 2,000 apps and run BIDTEXT on these apps. Then we inspect the results to discover popular API functions and create models only for those functions. These models are later used in the larger scale study.

During the pilot study, we also observe a kind of false positive that appears frequently. It is caused by a Facebook library used by many apps. The library logs an error message when it fails to obtain the device Id. The code snippet is abstracted as follows.

```
try {
  /* acquire device id */
} catch (Exception e) {
  Utility.logd("android_id", e);
}
```

The message `e` is typed with "*android_id*", which is a sensitive keyword. But the meaning of this message is indeed that the action of acquiring the device Id fails. Solving this issue requires in-depth semantic analysis of `e` which is not supported by BIDTEXT. Since the pattern is fixed, we post-process all the reports to filter out this pattern for both the pilot study and the later large scale study.

3.5.2 Unification vs. Bi-directional Propagation

In classic type inference, given an assignment statement such as `z=x+y` and `z=`ϕ`(x,y)`, the updated type sets of `x`, `y`, and `z` are the union of all three original type sets. In Section 3.3.2 (Rules Binary-Assignment and Phi-Assignment in Figure 3.4), we mentioned that such a unification based approach may produce a lot of false positives and hence BIDTEXT makes use of a bi-directional propagation strategy that avoids propagating type sets between right-hand-side operands (*i.e.,* `x` and `y` in the example). In this experiment, we want to compare these two propagation strategies.

Due to the lack of ground truth, such a study requires manually inspecting the reported disclosure defects and determining if they are false positives. Among the 2000 apps tested in the pilot study, we selected the first 60 apps whose *data disclosure path* (*i.e.,* the data flow subgraph that includes the path from the source to the sink and the path that the sensitive text is propagated from its origin to the sink) involves ϕ statements and/or binary operations with the unification based propagation policy. We re-run BIDTEXT on the 60 apps with the bi-directional propagation policy and compare the two sets of results.

Among these 60 apps, 42 of them are reported by both the unification policy and the bi-directional policy; 25 of them contains flows only reported by the unification policy. Note that the two do not add up to 60 because some apps have multiple reported disclosures, some being reported by both policies and the others being only reported by the unification policy. We manually studied the 25 cases reported by the unification policy and found that they are all false positives. We have shown one sample false positive in Section 3.3.2.

3.5.3 Large Scale Evaluation

In this experiment, we use 10,000 apps not covered by the pilot study. The apps have a minimum size of 6.46KB for the APK files and a maximum size of 49.94MB. The average size of the APK files is 9.17MB. Among these apps, there are two that do not contain any DEX bytecode in the APK files. For the remaining apps, the minimum size of the bytecode files (`classes.dex`) is 452 bytes and the maximum size is 10.32MB. The average size of the bytecode files is 2.53MB.

Results

The total analysis time for the 10,000 apps is 587.6 hours. Figure 3.10 presents the distribution of the cumulative analysis time for all the 10,000 apps. We divide the total analysis time into three parts according to how the analysis on an app terminates. As

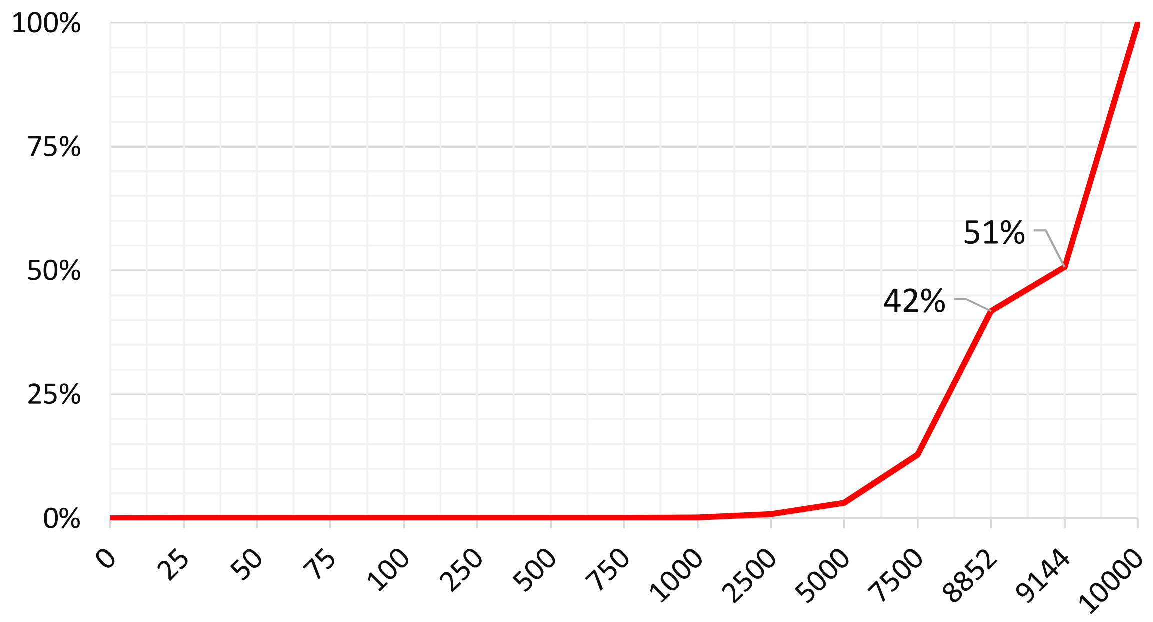


Figure 3.10.: Distribution of accumulative analysis time for all apps.

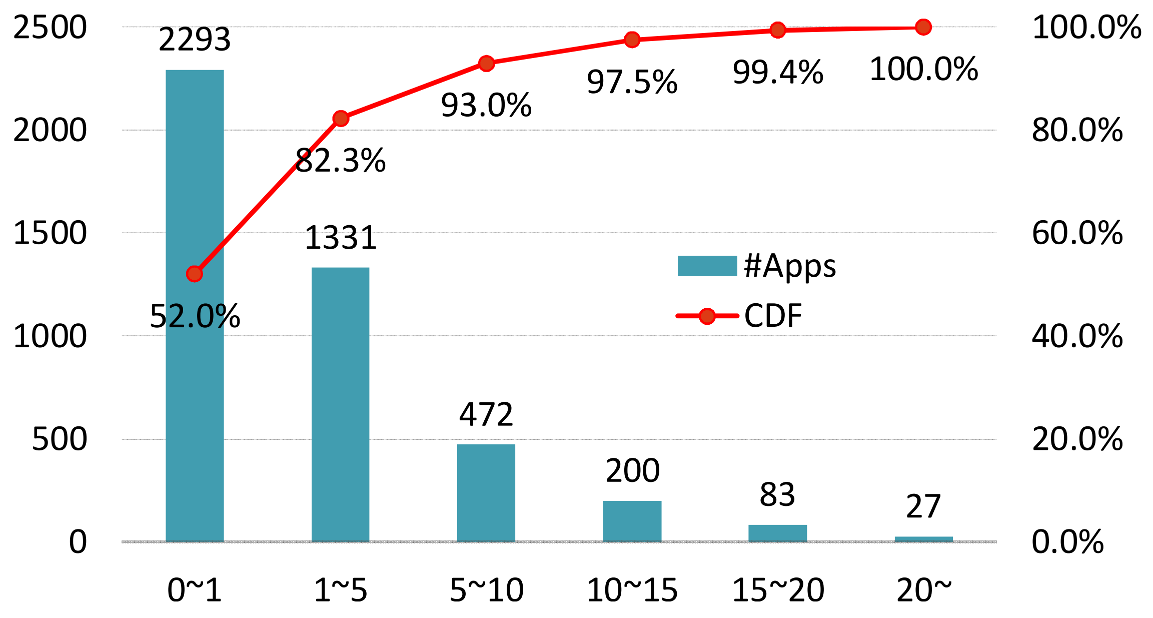
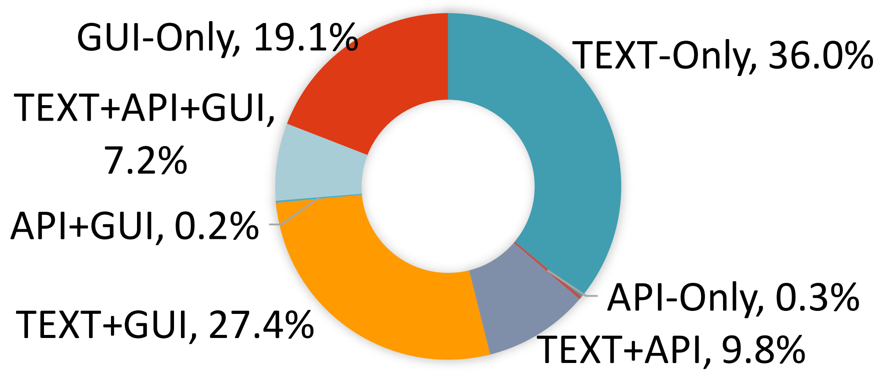
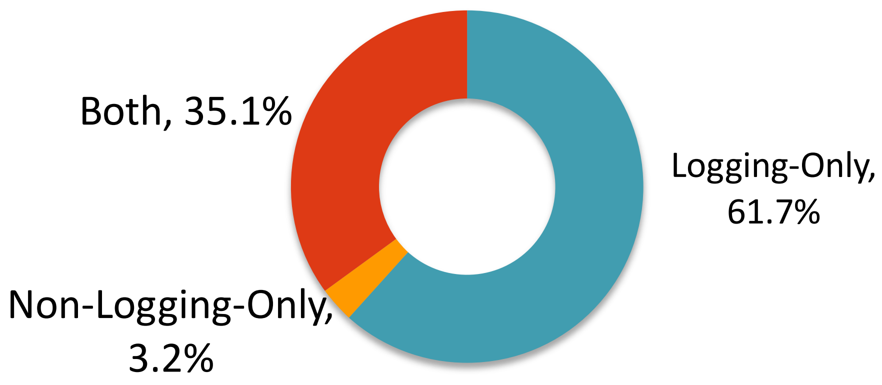


Figure 3.11.: Distribution for the analysis time (in minutes) of the apps reported with sensitive data disclosures.



(a) By sources.

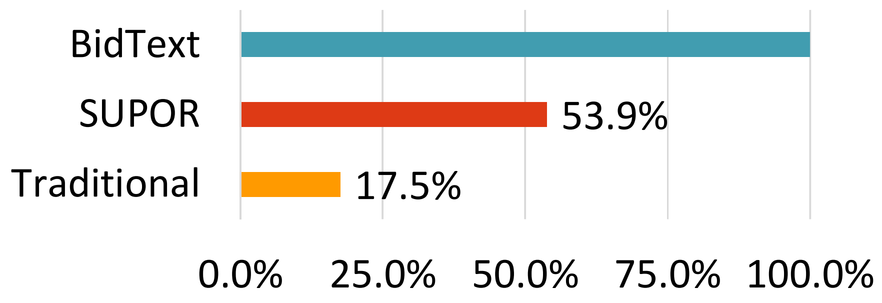


(b) By sinks.

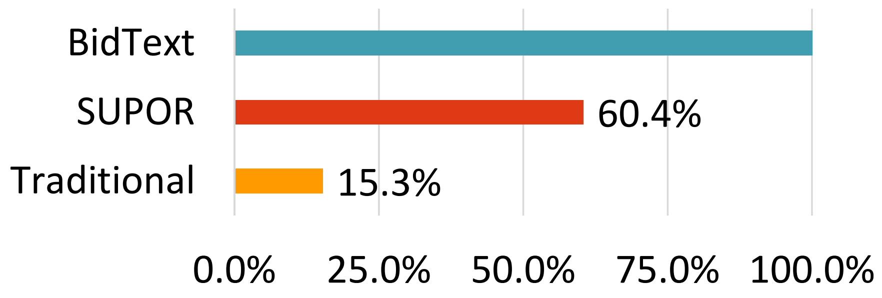
Figure 3.12.: Breakdown of the reported apps.

mentioned above, we set the analysis timeout to 20 minutes for each app. In our evaluation, 856 apps (8.56%) time out and the total analysis time account for 49% of the total time consumed for the 10,000 apps. We have 293 other apps of which the analysis ran out of memory. The total time for these apps accounts for 9%. For the remaining 8,852 apps that finished normally take only 42% of the total analysis time. Observe in Figure 3.10 that the first 7,500 apps take less than 15% of the total time. Among the 8,852 apps, the minimum analysis time is 0.2 seconds and the maximum time is 1197.4 seconds. The median is 24.9 seconds while the average time is 99.9 seconds. The largest app that terminates normally has the APK size of 49.94MB, and the bytecode size of 10.32MB.

Overall, BIDTEXT reports 4,406 apps with sensitive data disclosure problems. We show the analysis time distribution of these apps in Figure 3.11. The blue bars show the



(a) All sinks.



(b) Non-logging sinks.

Figure 3.13.: Comparing BIDTEXT with static tainting (tracking specific APIs) and SUPOR [59].

number of apps that finished within a time period. For instance, 472 apps took more than 5 minutes but less than 10 minutes. We also see that 27 apps timed out in the experiments, although partial results were collected before the analysis terminated. The red line presents the cumulative analysis time: 93.0% of the apps were analyzed within 10 minutes. We can conclude that BIDTEXT is efficient to be applied to market-scaled apps.

We also show the breakdown of the 4,406 apps by the sources of data disclosures in Figure 3.12a.

There are three types of sources: (1) TEXT – constant texts in the code that denote sensitive data; (2) API – sensitive API (recall that BIDTEXT also detects data disclosures originating from sensitive APIs by associated artificial texts to the source APIs such as `Location.getLatitude()`); and (3) UI – constant texts retrieved from user interfaces that denote sensitive data. Observe that the majority of disclosures are/can be detected by the sensitive text labels. Some data disclosure defects can be recognized through multiple sources (*e.g.,* TEXT+API), meaning that there are some (bi-directional) data flow paths from a sensitive API to a sink and from some constant text to the same sink. Consider the following example. The data flow path 2→6→7 denotes a disclosure originating from TEXT (*i.e.,* "*android_id*") and the path 4→6→7 denotes a disclosure originating from API (*i.e.,* "`getDeviceId()`").

```
1 if (fails_to_obtain_imei()) {
2   id = Settings.Secure.getString(resolver, "android_id");
3 } else {
4   id = telephonyManager.getDeviceId();
5 }
6 json.putString("id", id);
7 http_sink(json.toString()); // sink
```

The breakdown of the apps by the sink types is shown in Figure 3.12b. Note that 64.9% of the reported apps contain disclosures due to logging. Although data disclosure through logging is substantially mitigated by access control in the latest version of Android, it is still a security concern for legacy Android systems such that most existing works [9,35,36,59] report these disclosures. About 38.3% of the reported apps (16.9% of all the apps evaluated) contain sensitive data disclosures due to to non-logging sinks. They are serious threats even in the latest Android systems.

Figure 3.13 shows how BidText compares with an implementation of the traditional taint tracking technique (tracking disclosures from source APIs through forward data-flow similar to [4]) and SUPOR [59], which is a technique that tracks disclosures from sensitive UI elements (*e.g.,* input boxes) through forward data-flow. BidText always reports a super-set of those reported by the classic tainting and SUPOR. In the figure, the numbers of apps reported by tainting and SUPOR are normalized to those reported by BidText. Observe that they only report 17.5% and 53.9% of those reported by BidText, respectively. Even combining the two can only detect 64.0%. If only taking non-logging disclosures into account, they report 15.3% and 60.4% of those reported by BidText. This attributes to both the new text label correlation analysis and the bi-directional type set propagation strategy.

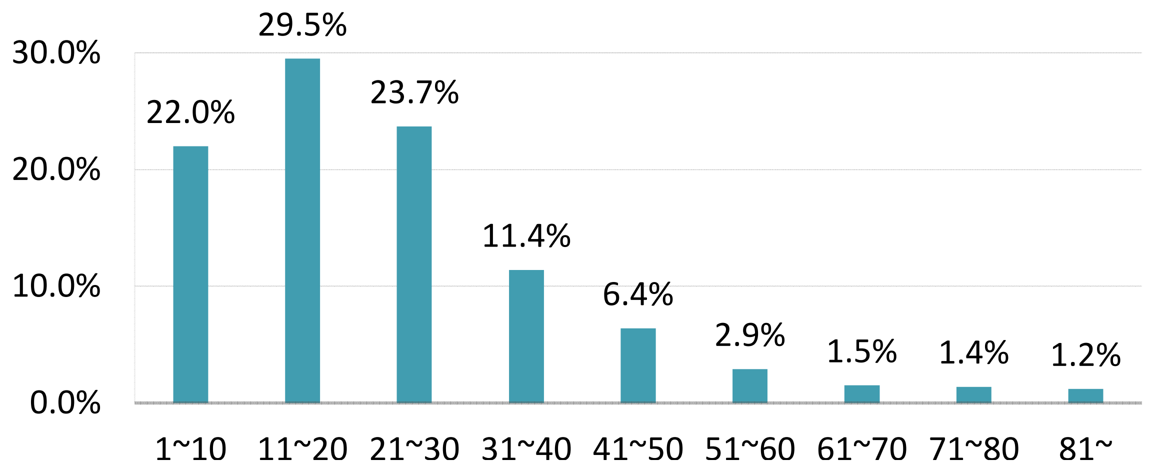


Figure 3.14.: Length distribution of the emitted paths for the reported apps. X-axis shows the length of the paths.

We present the length distribution of the emitted data disclosure paths for the 4,406 apps in Figure 3.14. Though some paths tend to be very long (more than 80 elements), most of them are relatively short. More than 75% of the paths require less than 30 steps from the origination of the sensitive texts to the sink points.

False Positives and False Negatives. It is critical to understand the quality of the reported defects. Due to the lack of ground truth, we had to perform manual inspection. Studying the full set of results is infeasible. Hence, we randomly chose 100 reported apps with a uniform size distribution for manual inspection. The results are presented in Table 3.1.

The columns indicate the sources of the disclosures. Row *Total* shows the total number of reported apps for each sources. Row *Only* shows the number of apps that only have reported disclosures falling into one category. The last row shows the number of false positives.

Observe that the 10 false positives are exclusive. Therefore, the false positive rate is 10%. The causes for false positives will be discussed in Section 3.5.3. We do not count the false negatives because we don't have the ground truth.

Among the 84 apps where disclosures are reported by code text analysis, 62 apps contain paths that can be only detected by our approach via text correlation analysis, *i.e.,* the data used at sink points neither come from any UI inputs nor from traditional source APIs.

Table 3.1.: Manually inspected evaluation results for 100 apps.

	TEXT	**API**	**UI**
Total	84	22	39
Only	44	2	14
FP	3	0	7

In other words, 62 of them cannot be detected by classic tainting or SUPOR. This ratio is consistent with that in Figure 3.13 for the larger experiment. The other reported disclosures have the sensitive data coming from these two categories of sources. They are reported by both BIDTEXT and the existing technique(s). Another interesting finding is that BIDTEXT often produces a shorter disclosure path. A typical scenario is that there is a long data flow path from a UI input element to a sink. However, mid way through the path, the (sensitive) data is put/get to/from some container with a sensitive textual key, which allows BIDTEXT to report a shorter path from the put/get operation to the sink. The benefits of shorter paths are two-folded: less human efforts needed for inspection and detecting more disclosures (because the full path from the source points to the sink points might be complicated, involving inter-component communications, such that the tool may fail to traverse the full path).

Case Studies

We observe many cases in which sensitive textual keys appear together with data in key-value operations, *e.g.,* constructing a name value pair (*e.g.,com.gunsound.eddy.fafapro*), inserting data into a hash map (*e.g.,me.tango.fishepic*), retrieving/adding data to persistent storage through an instance of `SharedPreferences` (*e.g., com.ifreeindia.sms_mazaa*) or putting data into a JSON object (*e.g., com.mobilegustro.war.battle.air.force*). BIDTEXT recognizes the sensitiveness of corresponding data via text correlation analysis.

In the following, we show a code snippet adopted from app *com.-pro.find.differences* that discloses sensitive device information to Web servers.

```
 1 void obtainDeviceInfo() {
 2   TCore.aid = Settings.Secure.getString(resolver,
         "android_id");
 3 }
 4 void connectWebServer() {
 5   Map map = new HashMap();
 6   safePut(map, "android_id", TCore.aid);
 7   String params = convertURLParams(map); // omitted
 8   http_sink(params); // sink
 9 }
10 void safePut(Map map, String k, String v) {
11   map.put(k, v);
12 }
```

The method call at line 2 returns system information based on the given key value. For example, a unique Id for the device is obtained if “*android_id*” is given as the key. If the key is “*enabled_input_methods*”, the return value contains a list of input methods that are currently enabled. Therefore, the sensitiveness of the return value depends on the key. BIDTEXT works by correlating the textual key with the return variable to decide whether a later sink operation involves sensitive data or not.

In the above example, the variable `TCore.aid` is typed with the constant text “*android_id*” at line 2, which is later propagated to parameter `v` of method `safePut()` at line 10. `v` is inserted into the hash map at line 11. Note that “*android_id*” at line 6 is propagated to `k`@10 which is further propagated to the hash map and variable `v` according to the corresponding API model for propagation. Along the data flow, the constant text is propagated to `params`@7 that is eventually used at the sink point at line 8. BIDTEXT reports the data disclosure.

False Positives

One of the 10 false positives is caused by unmodeled API functions. The corresponding code snippet is from app *at.zuggabecka.-radiofm4*.

```
1 uidx = cursor.getColumnIndex("username");
2 iidx = cursor.getColumnIndex("_id");
3 id = cursor.getLong(iidx);
4 sink(id);
```

At line 1, a sensitive keyword "*username*" is correlated with the receiver object `cursor` that is related to a database query. Then all uses of `cursor` propagate the text label to other variables, *e.g.,* the return value of a relevant method call. Thus, `id` at line 3 is typed with "*username*". Later when it is used at a sink point, BIDTEXT reports a sensitive data disclosure after analyzing the corresponding type set. To remove this false alarm, we can build a model for API `Cursor.getColumnIndex(key)` to only propagate type set from `key` to the return value, avoiding propagating to the receiver object. Then in the above code snippet, only variable `uidx@1` is typed with "*username*". Variable `id` that appears at the sink point is only typed with "*_id*" which is not considered as a sensitive keyword. Therefore there is no disclosure problem with the model.

All the other nine false positives are caused by incorrect recognition of text, two for code text and seven for UI text.

App *com.netcosports.andalpineski* contains a text label "*Apps-_lang[apps_lng_iso2]*" which indicates the language of the app. However, it contains a predefined sensitive keyword "*lng*" which is mostly used as an abbreviation of "*longitude*". Failing to understand the meaning of the text, BIDTEXT incorrectly reports a sensitive data disclosure.

App *com.wactiveportsmouthcollege* has a UI text of "*Pin to desktop*" where sensitive keyword "*Pin*" is used as a verb. Failing to understand it leads to a false positive. All other false positives have similar causes – sensitive keywords in a phrase or sentence do not indicate any sensitive information. Possible solutions for this type of false positives include integrating more advanced NLP techniques with program analysis to understand the meanings of the text.

3.5.4 Discussion

One limitation of BIDTEXT lies in that the text in code may not be in a generalized format. For example, some developers use "*lng*" for "*longitude*" whereas others use "*long*" for it, which is a more general word in English. If we treat "*long*" as a sensitive keyword, we can expect many false positives. In addition, developers tend to combine several words (or abbreviations) into a single word, which makes it more difficult to determine whether the correlated data are sensitive or not.

In the future, we plan to improve our approach in the following aspects. The first one is to discover text labels in the names of method calls, if they are not obfuscated, and variable/field names. The second improvement is to consider code comments if source code is available. The third one is to improve the NLP aspect by putting the keywords in their program context. Doing so, we may be able to recognize "*long*" indeed means longitude.

3.6 Related Work

A lot of prior research has focused on detecting sensitive data disclosures, either statically or dynamically, for mobile apps [1, 4, 31, 35, 36, 64]. Most of them consider specific APIs as sensitive source points while BIDTEXT analyzes text labels to determine if a variable can hold sensitive data. SUSI [37] gives a comprehensive list of the data sources in Android, but it does not assume the data obtained from the sources must be sensitive. In addition, even if the state-of-the-art static detectors, *e.g.,* FlowDroid [35, 36] and DroidSafe [64], had been enhanced with various ways of determining data sensitiveness, they would likely not be able to detect some sensitive data disclosures reported by BIDTEXT such as our motivating example, where the sensitiveness of the data is determined after the sink point and there is no forward data-flow from the sensitiveness revelation point and the sink point. BIDTEXT, however, leverages bi-directional propagation to address this problem.

Huang *et al.* developed type-based taint analysis to detect information leaks in Java-based Web applications and Android apps via type inference [65, 66]. They abstract the information flow analysis into a type system and check if any type error occurs. Their technique scales well without using advanced points-to analysis [35, 36, 64]. Their technique still follows the traditional definition of data disclosure, which is a forward data flow path from the source to the sink. In other words, it does not propagate data sensitiveness in a backward fashion. As such, it may not be able to report many disclosures reported by BIDTEXT, including the motivating example. Furthermore, their type system does not leverage text information. Ernst *et al.* also developed a type-based taint analysis system [67]. Their technique associates a few (security) types such as LOCATION, INTERNET, and SMS to sources and sinks and have a set of predefined policies such as LOCATION can only be compatible, or type-checked, with INTERNET. So if LOCATION reaches a program point with the SMS type, a leak is reported. Their flow analysis is forward whereas BIDTEXT is bi-directional. And BIDTEXT leverages text labels.

SUPOR [59] and UIPicker [43] discover sensitive information on user interface through static analysis. However, they essentially belong to the traditional forward data-flow based techniques. AsDroid [40] collects the set of API calls in an event handler and compares the meaning of these API calls with the UI text of the event to detect unwanted/unexpected app behavior. In contrast, BIDTEXT types individual variables in the program with text labels and leverages a type system that allows bi-directional propagation. Researchers also combine code and comment analysis to detect bugs or inconsistencies [68–70]. We envision comment analysis can leverage our bi-directional type system so that the information in comments can be leveraged to analyze fine-grained and in-depth app behavior. In addition, WHYPER [7] and AutoCog [8] apply NLP techniques to app's descriptions to obtain a comprehensive view of the app and check if the required permissions are appropriately specified in the descriptions. Besides, [71] and [72] apply NLP techniques on API descriptions or documents to infer method specifications. We can leverage these techniques to automate the generation of API models used in BIDTEXT.

3.7 Summary

We propose BIDTEXT, a novel static technique to detect sensitive data disclosures. BIDTEXT identifies text labels appearing in both code and UI, treats them as types, associates them to the corresponding variables, bi-directionally propagates the types through data flow and eventually attributes them to sink points that potentially disclose sensitive information. At the end, the parameters at the sink points have type sets of correlated texts. Textual analysis is applied to the type sets to determine if the variables may hold sensitive data. We implement BIDTEXT and preliminarily evaluate it on 10,000 apps downloaded from Google Play store. The preliminary results show the false positive rate is 10%.

4 ASDROID: DETECTING STEALTHY BEHAVIORS IN ANDROID APPLICATIONS BY USER INTERFACE AND PROGRAM BEHAVIOR CONTRADICTION

Android smartphones are becoming increasingly popular. The open nature of Android allows users to install miscellaneous applications, including the malicious ones, from third-party marketplaces without rigorous sanity checks. A large portion of existing malwares perform stealthy operations such as sending short messages, making phone calls and HTTP connections, and installing additional malicious components. In this paper, we propose a novel technique to detect such stealthy behavior. We model stealthy behavior as the program behavior that mismatches with user interface, which denotes the user's expectation of program behavior. We use static program analysis to attribute a top level function that is usually a user interaction function with the behavior it performs. Then we analyze the text extracted from the user interface component associated with the top level function. Semantic mismatch of the two indicates stealthy behavior. To evaluate AsDroid, we download a pool of 182 apps that are potentially problematic by looking at their permissions. Among the 182 apps, AsDroid reports stealthy behaviors in 113 apps, with 28 false positives and 11 false negatives.

4.1 Introduction

Android smartphones are becoming increasingly popular. Gartner's analysis shows that 72.4% of smartphones are based on Android [73]. A prominent characteristic of Android phones is that users can easily install miscellaneous apps downloaded from third-party marketplaces without jail-breaking. However, the downside is that Google and other vendors can hardly control the quality of apps on third-party marketplaces. Adversaries can submit their malicious apps and tempt users to install with various lures. Juniper Networks Mobile

Threat Center reported a dramatic growth in Android malware population from roughly 400 samples in June 2011 [74] to 175,000 in the third quarter of 2012 [75]. Most are present on third-party marketplaces.

A very popular category of Android malware features steal-thy malicious operations such as making phone calls, sending SMS messages to premium-rate numbers, making undesirable HTTP connections and installing other malicious components. It was reported by three recent studies [19,76,77] that 52-64% of existing malwares send stealthy premium-rate SMS messages or make phone calls. Note that these actions cause unexpected charges to phone bills [78, 79]. It was observed that stealthy HTTP requests are also very common undesirable behavior in malwares [76]. Besides leaking user information, they could also cause unexpected data plan consumption. In China, it was reported in March 2012 that more than 210,000 Chinese mobile devices were affected by a kind of malwares that could make stealthy HTTP connections inducing charges. They caused around 8 million dollars loss [80].

Despite the pressing need, detecting such malware is challenging as the malicious behavior appears to be indistinguishable from that of benign apps. For example, an online shopping app usually provides operation interfaces to help users conveniently call a service number or send a query SMS message. Apps providing travel-aid and adult content often allow users to make phone calls or send messages. Many benign apps allow establishing background HTTP connections (e.g. weather, stock trading and gaming apps). Many also allow users to install additional components.

Existing techniques are insufficient in detecting/preventing stealthy malicious behaviors. A very important protection mechanism on Android is to allow users to perform access control by setting application privileges. However, the access control is very coarse-grained. For example, the SMS messaging capability can either be enabled or completely disabled. It is hard to decide if we should disable for a given app as many benign apps do send SMS messages. Taint analysis [1, 4, 35, 36] allows detecting information leak in apps. But the stealthy behavior in malwares may not leak any private information. Recently, Google provides the capability of blacklisting certain premium-rate phone numbers [81], which provides a potential way of preventing stealthy SMS messages or phone calls. However, keeping such a blacklist up-to-date is a non-trivial challenge. In some countries such as China, there is no difference between a premium-rate number and a regular phone number.

In this chapter, we propose a novel technique to detect stealthy malicious behaviors in Android apps. We model stealthy behavior as *the program behavior mismatches with user interface*. The intuition is that user interface (UI) represents the user's expectation of program behavior. Hence, it can naturally serve as an oracle to detect behind-the-scene behavior. For example, an SMS message send triggered by a user interaction that is supposed to set the background color should be considered malicious. The technique consists of two components. One is the static program analysis component that attributes the behavior of interest (e.g. SMS send and HTTP connection) to a top level function with associated UI (e.g. the `onClick()` function of a button). The other is the UI analysis component that makes use of text analysis to analyze the intent described by the corresponding interface artifacts (e.g. the text associated with the button). Any mismatch will be reported as potentially malicious. In the program analysis component, we classify Android APIs into different groups. Each group is assigned an intent type such as SMS send and phone calls. Reachability analysis is performed on control flow graph (CFG) and call graph (CG) to propagate such intents from the API call sites to top level functions. Note that in event driven programming, an invocation of a top level function usually denotes an action or a task that can be considered as a natural unit to reason about stealthiness. The interface analysis component identifies the text of the UI artifact associated with a top level function. Then compatibility check is performed between the intents from program analysis and those extracted from the interface text.

Our contributions are summarized as follows.

- We propose a method to detect Android malware that performs stealthy operations including SMS message send, phone calls, HTTP connections and component installations. It is based on the novel idea of detecting mismatches between program behavior and user interface.

- We found that in many cases even though there is no direct match between an API intent (e.g. SMS send) and the UI text, the API may be correlated with other APIs that explicitly expose the behavior (e.g. an API call that logs the SMS send to the mail box). In such cases, the behavior should not be considered stealthy. We propose an in-depth analysis that considers program dependences between APIs to identify their correlations and hence improve precision.

- We formally present our design using *datalog* rules. The design handles a number of Android-specific challenges.

- We implement a prototype called AsDroid (*Anti-Stealth Droid*). We collect a pool of 182 apps that have the permissions to perform the malicious operations of interest. AsDroid reports that 113 of them have stealthy behaviors, with 28 false positives and 11 false negatives.

4.2 Motivating Example

We use a real application *Qiyu* to motivate our technique. It is a location-based social networking service application on Android. Some relevant code snippets are shown in Figure 4.1 and part of the corresponding call graph is in Figure 4.2. The entry function `onClick()` (at line 2) is the handler of a button with text "`One-Click Register & Login`". The scenario is as follows. When the user clicks the button, the app checks the current environmental settings. In most cases, the true branch is taken, in which an asynchronous task is appended to the task queue and executed (line 5). This causes an indirect invocation to a predefined handler `doInBackground()` at line 11, which is always implicitly called by the Android runtime to perform some background processing when a task starts to execute. The function transitively calls method `A()` (in class `Woa.BA`) at line 17. The method connects to a website through `HttpClient.execute()` at line 18 to perform registration or login. The chain of function calls is also shown on the left of Figure 4.2. When the test at line 3 fails, the else branch (line 6) is taken. A different

```
 1 // In class Qiyu.StartPageActivity
 2 public void onClick(View v){
 3   if(/*test environment*/){
 4     Woa.F f = new Woa.F(v, this);
 5     f.execute(new String[0]);//trigger line 11
 6   } else ...{
 7     Woa.AG.B();//invoke line 21
 8   }
 9 }
10 // In class Woa.F
11 public Object doInBackground(Object[] objs){
12   //transitively calls Woa.BA.A() at line 17
13 }
14 // In class Woa.BA
15 private org.apache.http.client.HttpClient h;
16 private org.apache.http.client.methods.HttpGet d;
17 public void A(){
18   this.h.execute(this.d); //HttpClient.execute(...)
19 }
20 // In class Woa.AG
21 public static void B(){
22   Woa.U u = new Woa.U();
23   u.execute(...);//transitively calls C() at line 26
24 }
25 // In class Woa.AK
26 public static boolean C(Context c, String s1, String s2){
27   SmsManager sm = SmsManager.getDefault();
28   sm.sendTextMessage(s1, null, s2, null, null);
29 }
```

Figure 4.1.: Simplified code snippet for app Qiyu.

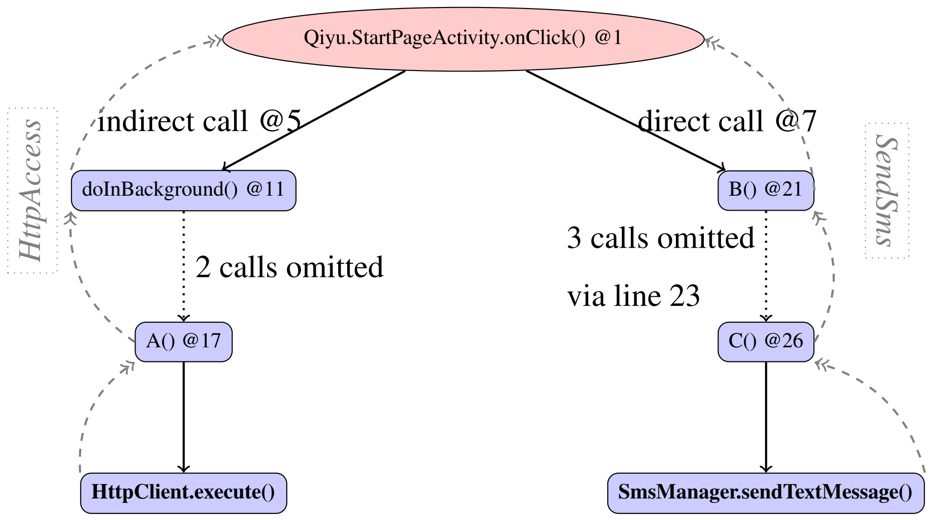


Figure 4.2.: Call graph and intent propagation in app Qiyu.

chain of function invocations are made, eventually leading to an SMS message being sent inside method `C()` (in class `Woa.AK`) at line 28 without the user's awareness. The chain is shown on the right of Figure 4.2. Note that we omit three function calls between the asynchronous task execution at line 23 and method `C()` for brevity.

To detect stealthy behaviors, our program analysis component first attributes top level functions with intents by analyzing the operations of interest directly or transitively performed by such functions. We classify Android APIs to a few pre-defined intent types. In this example, `HttpClient.execute()` at line 18 denotes the **HttpAccess** intent and `SmsManager.sendTextMessage()` at line 28 denotes the **SendSms** intent. The intents get propagated upward along the call edges (see Figure 4.2) and eventually aggregated on the top level node `onClick()`, which is a user interaction function, suggesting the operations performed by this function should reflect what the UI states. The UI analysis component identifies the UI artifacts corresponding to the `onClick()` function, i.e. the button and its residence dialog. It further extracts the text on these interface artifacts and performs text analysis to identify a set of keywords. In this example, they are "`Register`" and "`Login`". AsDroid looks-up the compatibility of the keywords and the intents identified by the program analysis component from a dictionary generated before-hand in a training phase. In this case, the **HttpAccess** intent is compatible but **SendSms** is not. Our tool hence reports the contradiction.

There are cases that multiple intents of a top level function are correlated. For example, a dialog may be popped up after a SMS message send to indicate the success of the send, even though the button that initiates the send does not have any textual hint about sending messages. In this case, the SMS send is not stealthy. The display of a dialog has the **UiOperation** intent. Both the **UiOperation** and **SendSms** intents reach the top level function. We hence analyze if the intents are correlated by analyzing their program dependences. Since **UiOperation** is not stealthy, the correlation between the **UiOperation** and **SendSms** intents suggests the sanity of the SMS send behavior.

4.3 Design

In this section, we first define six types of intents that are of our interest. The corresponding APIs are commonly used in Android apps.

SendSms. This intent corresponds to SMS send APIs, including `sendTextMessage()`, `sendDataMessage()` and `sendMultipartTextMessage()` declared in the class `SmsManager`. These API functions are usually executed in the background. An SMS send through a separated messaging app is not taken into consideration in this research because it requires the user to explicitly interact with the messaging app to finish the process and hence is not stealthy.

PhoneCall. It corresponds to a direct phone call, namely, invoking `startActivity()` with action `android.intent.action.CALL`. Malware can leverage the automated calling mechanism to dial a number without the user's awareness. Phone calls can also be made through `startActivity()` with an action `android.intent.action.DIAL`. However, we do not model this API because explicit user approval is needed when the API is used.

HttpAccess. This intent describes HTTP access APIs. It includes `URL.openStream()`, `URL.openConnection()`, `AbstractHttpClient.execute()`, *etc.*. HTTP access is commonly used in Android apps for a wide range of purposes.

Install. It describes API functions that are for installing other components or applications. Many Android malwares have their payload as installing another piece of malicious code. Benign apps may also need to perform installation, which is however usually authorized or explicitly guided by the user. Modeled functions include `Runtime.exec()` with `"pm install"` as the argument, and `ProcessBuilder.start()` using `"pm"` and `"install"` to build a new process.

SmsNotify. In some cases, the user does not need to (or cannot) authorize a message send operation. But after the operation, the app may automatically notify the user that there was an SMS send. In this case, we should not consider the message send as a stealthy action even though the user interface that leads to the SMS send operation does not have any textual implication of the operation. One typical example is that a copy of the message is saved to the user's mail-box to record what just happened. Hence, we model the following API to the **SmsNotify** intent: `ContentResolver.insert()` and the destination table is given by a URL "`content://sms`". It means inserting data into the preloaded database for short messages.

UiOperation. A top level user interaction function may display more user interface elements to allow further interactions with the user. In some cases, UI display operations may be correlated to some of the aforementioned intents. For example, a dialog may be popped up after an SMS send to notify the user about the send. In such cases, the SMS send is not stealthy. To reason about these cases, we associate the UI display API functions such as `ImageView.setImageBitmap()`, `View.setBackgroundDrawable()` and `AlertDialog$Builder.setMessage()`, with the **UiOperation** intent.

4.3.1 Intent Propagation

In this section, we describe how intents are propagated to top level functions such that we can check compatibility with the corresponding UI text. We also describe how to detect correlation between intents. Intent propagation is based on call graph. The calling convention of Android apps has its unique features, which need to be properly handled. Intent correlation analysis is mainly based on program dependences. However, correlated intents do not simply mean there are (transitive) dependences between them.

The analysis is formally described in the *datalog* language [82], which is a Prolog-like notation for relation computation. It provides a representation for data flow analysis in the form of formulated relations. The inference rules on these relations are shown in Figure 4.3 and Figure 4.4. Relations are in the form $p(X_1, X_2, ..., X_n)$ with p being a predicate. X_1, X_2, ..., X_n are terms of variables or constants. In our context, variables are essentially program artifacts such as statements, program variables and function calls. A predicate is a declarative statement on the variables. For example, *inFunction*(F,L) denotes if a statement with label L is in function F.

Rules express logic inferences with the following form.

$$H \textbf{ :- } B_1 \textbf{ \& } B_2 \textbf{ \& ... \& } B_n$$

H and $B_1, B_2,...B_n$ are either relations or negated relations. We should read the :- symbol as “if”. The meaning of a rule is if B_1, B_2,...B_n are true then H is true.

Relations can be either inferred or atoms. We often start with a set of atoms that are basic facts derived from the compiler and then infer the other more interesting relations through our analysis. We use WALA [28] as the underlying analysis infrastructure. We leverage its *single static assignment* (SSA) representation, control flow graph, part of call graph, and the MAY-points-to analysis to provide the atoms.

Atom *apiIntent*(L,T) denotes an intent T is associated with an API call at L, reflecting our API classification. Atom *hasDefFreePath*(L_1,L_2,X) indicates there is a program path from program point L_1 to L_2 and along the path (not including L_1 or L_2), variable X may not be defined. This is to compute the *defUse*(L_1, L_2) relation that denotes if a variable is

$apiIntent(L,T)$	:	API call at program point L has intent type T.
$def(L,X)$	:	variable X is defined at program point L.
$use(L,X)$	:	variable X is used at program point L.
$actual(L,M,X)$	:	variable X is the M^{th} actual argument at call site L.
$formal(F,M,X)$	:	variable X is the M^{th} formal argument of function $F()$.
$inFunction(F,L)$	:	program point L is in function $F()$.
$funEntry(F,L)$	:	program point L is the entry of function $F()$.
$hasDefFreePath(L_1,L_2,X)$	:	there is a path from L_1 to L_2 along which X may not be defined.
$componentEntry(X,F)$	:	$F()$ is the entry of Android component X. e.g. *onCreate*() of an *Activity* or a *Service* component.
$immediateCD(L_1,L_2)$	:	program point L_2 is immediately control dependent on L_1 in the same function.
$directInvoke(F_1,F_2,L)$	:	F_1 invokes F_2 at program point L
$indirectInvoke(F_1,F_2)$	:	F_2 is the actual destination of $F_1()$ in event-driven circumstances, e.g. (1) *Thread.start*() $\rightarrow$ *Runnable.run*(); (2) *Handler.sendMessage*() $\rightarrow$ *Handler.handleMessage*().
$iccInvoke(F_1,F_2,L)$	:	F_1 invokes a function F_2 for inter-component communication purpose at L. F_2 should be APIs like *startActivity*(), *startService*().

Figure 4.3.: Datalog atoms for intent propagation and correlation.

/*$invoke(F_1,F_2,L)$: F_1 invokes F_2 at program point L.*/

$invoke(F_1,F_2,L)$	:-	$directInvoke(F_1,F_2,L)$
$invoke(F_1,F_2,L)$	:-	$iccInvoke(F_1,F_3,L)$ & $actual(L,1,X)$ & "L_1: $X.setClass(...)$" & $actual(L_1,2,Y)$ & $componentEntry(Y,F_2)$
$invoke(F_1,F_2,L)$	:-	$invoke(F_1,F_3,L)$ & $indirectInvoke(F_3,F_2)$
$invoke(F_1,F_2,L)$	:-	$invoke(F_1,F_3,L)$ & $invoke(F_3,F_2,L)$

/*$hasIntent(F,T,L)$: $F()$ has *intent* type T and the corresponding API call is at L.*/

$hasIntent(F,T,L)$	:-	$invoke(F,A,L)$ & $apiIntent(L,T)$
$hasIntent(F,T,L_1)$	:-	$hasIntent(F_1,T,L_1)$ & $invoke(F,F_1,L_2)$

/*$controlDep(L_1,L_2)$: program point L_2 is control dependent on L_1.*/

$controlDep(L_1,L_2)$	:-	$immediateCD(L_1,L_2)$
$controlDep(L_1,L_2)$	:-	$inFunction(F_1,L_1)$ & $inFunction(F_2,L_2)$ & $invoke(F_1,F_2,L_3)$ & $controlDep(L_1,L_3)$

/*$defUse(L_1,L_2)$, $useUse(L_1,L_2)$: data at L_1 and L_2 are data correlated.*/

$defUse(L_1,L_2)$	:-	$def(L_1,X)$ & $use(L_2,X)$ & $hasDefFreePath(L_1,L_2,X)$
$defUse(L_1,L_2)$	:-	$invoke(F_1,F_2,L_1)$ & $actual(L_1,M,X)$ & $formal(F_2,M,Y)$ & $funEntry(F_2,L_3)$ & $hasDefFreePath(L_3,L_2,Y)$ & $use(L_2,Y)$
$useUse(L_1,L_2)$	:-	$defUse(L_3,L_1)$ & $defUse(L_3,L_2)$
$useUse(L_2,L_1)$	:-	$defUse(L_3,L_1)$ & $defUse(L_3,L_2)$

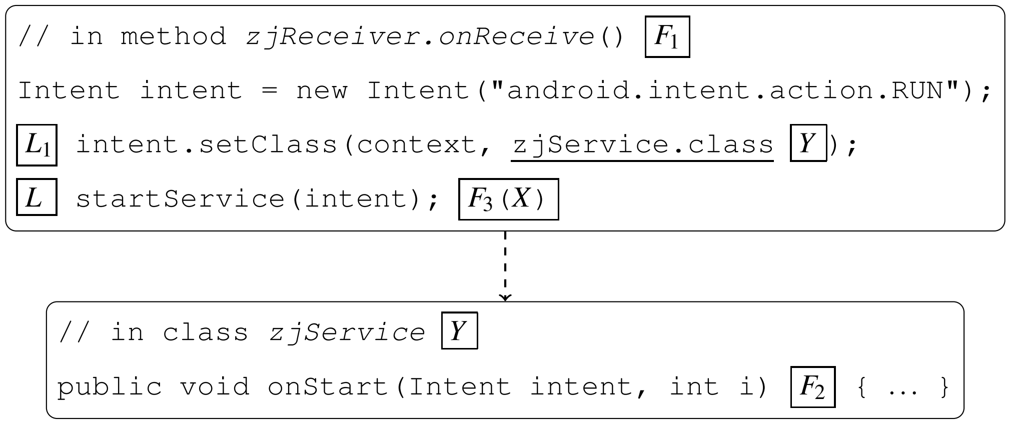
/*$correlated(L_1,L_2)$: L_1 and L_2 are data/control correlated.*/

$correlated(L_1,L_2)$	:-	$controlDep(L_1,L_2)$
$correlated(L_1,L_2)$	:-	$defUse(L_1,L_2)$
$correlated(L_1,L_2)$	:-	$useUse(L_1,L_2)$
$correlated(L_1,L_2)$	:-	$correlated(L_1,L_3)$ & $correlated(L_3,L_2)$

/*$correlatedIntent(F,T_1,L_1,T_2,L_2)$: In function F, intent T_1 at L_1 is correlated to T_2 at L_2*/

$correlatedIntent(F,T_1,L_1,T_2,L_2)$	:-	$hasIntent(F, T_1, L_1)$ & $hasIntent(F, T_2, L_2)$ & $correlated(L_1,L_2)$

Figure 4.4.: Datalog rules for intent propagation and correlations



```
// in method zjReceiver.onReceive() F1
Intent intent = new Intent("android.intent.action.RUN");
L1 intent.setClass(context, zjService.class Y);
L  startService(intent); F3(X)

// in class zjService Y
public void onStart(Intent intent, int i) F2 { ... }
```

Figure 4.5.: ICC call chain example in app GoldDream.

defined at L_1 and used at L_2. To generate the atom relation, we leverage the SSA form and the points-to analysis. The analysis is conservative. If we are not sure X must be re-defined along the path, we assume the path is definition free. The paths we are considering include both intra- and inter-procedural paths.

Android apps are component based. Generally, there are four types of basic components: *Activity*, *Service*, *Broadcast Receiver* and *Content Provider*. Activity component is for a single UI screen. Service component is for long-running operations in the background (without any UI). Broadcast receiver responds to system-wide broadcast announcements. Content provider is used for application data management [83]. Inter-Component Communication (ICC) is used to deliver data between components, which is similar to traditional function invocations. We have to model such communication as a function may transitively invoke API functions with intent of interest through ICC. However, the calling convention of ICC is so unique that the underlying WALA infrastructure cannot recognize ICC invocations. Figure 4.5 shows an example from a real world app *GoldDream*. Inside the `zjReceiver.onReceive()` function, there is an ICC call to the `onStart()` function of the `zjService` component. Observe that the invocation is performed by creating an Android `Intent` object[1], which can be considered as a request that gets sent to

[1]`Intent` is a standard class in Android. We call it Android Intent in order to distinguish with the intents we associate with API functions.

other components to perform certain actions. The target component is set by explicitly calling `setClass()` of the Android Intent object. The request is sent by calling `start-Service()` with the Android Intent object. The Android runtime properly forwards the request to the `onStart()` function of the `zjService` component.

To capture such call relation, we introduce the *componentEntry*(X,F) atom with X a subclass of `Service`, `Activity` or `BroadcastReceiver`. The entry point F denotes `onCreate()`, `onStart()`, and `onReceive()`, which are also called *lifecycle* methods by Android developers. We introduce atom *iccInvoke*(F_1,F_2,L) with F_2 denoting special ICC functions, such as `startActivity()`, `startService()` and `send-Broadcast()`. The second inference rule of the *invoke*(F_1, F_2,L) relation describes how we model ICC as a kind of function invocation. Let's use the example in Figure 4.5 to illustrate the rule. It allows us capture the call chain `zjReceiver.onReceive()` $\rightarrow$ `startService()` $\rightarrow$ `zjService. onStart()`. Labels $\boxed{L}$, $\boxed{L_1}$, $\boxed{F_1}$, $\boxed{F_3}$, and $\boxed{Y}$ in Figure 4.5 correspond to those in the second *invoke*() rule.

Atom *directInvoke*(F_1,F_2,L) denotes regular function calls including virtual calls, leveraging WALA. Atom *indirectInvoke*(F_1,F_2) denotes another special kind of function invocations in Android apps, namely, implicit calls in thread execution and event handling. A typical indirect call is a thread-related invocation, *e.g.,* the actual call destination of `Thread.start()` is the `run()` method of the corresponding class. The function call `f.execute()` $\rightarrow$ `doInBackground()` in Figure 4.1 (*i.e.,* line 5 $\rightarrow$ line 11) is an example for event handling indirect invocation. We detect these implicit calls through predefined patterns.

Relation *hasIntent*(F,T,L) denotes function F is tagged with an intent T initiated by the API call at program point L. For example, in Figure 4.1, we can infer the following:

hasIntent (F = `StartPageActivity.onClick()`,

T = **SendSms**,

23 /*`sm.sendTextMessage(...)`*/) = TRUE.

Observe that the first *hasIntent*() rule tags the enclosing function of an API call. The second rule propagates a tag from a callee to the caller. Note that a function may have multiple intents. These intents may be of the same type (but initiated at different API call locations).

The remaining relations and rules are for intent correlations. Relation *correlated*(L_1,L_2) determines if two program points L_1 and L_2 are correlated. Correlation can be induced by definition-use, use-use, and control dependence relations, described by relations *defUse*(), *useUse*(), and *controlDep*(), respectively. The fourth *correlated*() rule suggests that the relation is transitive.

The first rule of *defUse*(L_1,L_2) is standard. In our implementation, we leverage SSA form to derive definition-use relation for local and global variables. We leverage points-to relation to reason about definition-use relation for object fields. The second rule is to capture definition-use relation by parameter passing, including those through Android specific calling conventions. The basic idea is that we consider a formal argument Y used inside the callee at L_2 is defined at the call site L_1 (in the caller) if it is not re-defined along the path from the callee entry to the use site.

The relation *useUse*(L_1,L_2) denotes that there are uses at L_1 and L_2 coming from the same definition point. For example, L_1 and L_2 could be the two uses of the same variable in the two branches of a predicate. Considering use-use relation in the *correlated*() relation is the key difference from standard program dependence analysis that considers only definition-use and control dependence relations.

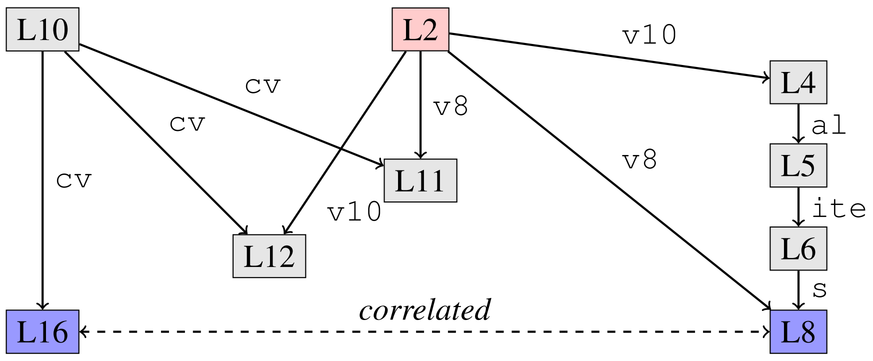
Computation of *controlDep*(L_1,L_2) is standard except that it also models inter-procedural control dependence. Particularly, all statements in a callee have control dependence with a predicate in the caller that guards the call site.

Finally, the relation *correlatedIntent*(F,T_1,L_1,T_2,L_2) denotes if two intents T_1 and T_2 at function F are correlated.

Example. Figure 4.6 shows a correlation analysis example in app *Shanghai 1930*. `ContentResolver.insert()` at line 16 stores the sent text message into the mail box and it hence has intent type **SmsNotify**. It is determined to be correlated to the SMS sending operation with **SendSms** intent at line 8. According to the definition-use graph in Figure 4.6(b), line 16 is correlated with line 11 (both use *cv* defined at line 10) by the *useUse*()

```
1  // in class PaySmsActivity
2  void a (String v8, String v9, String v10) {
3    SmsManager sm = SmsManager.getDefault();
4    ArrayList al = SmsManager.divideMessage(v10);
5    Iterator<String> ite = al.iterator();
6    while (ite.hasNext()) {
7      String s = ite.next();
8      sm.sendTextMessage(v8,v9,s,null,null);
9    }
10   ContentValues cv = new ContentValues();
11   cv.put("address",v8);
12   cv.put("body",v10);
13   cv.put("type",2);
14   ContentResolver cr = getContentResolver();
15   Uri uri = Uri.parse("content://sms");
16   cr.insert(uri,cv);
17 }
```

(a) Code snippet.



(b) Part of definition-use relations. Solid arrows labeled with variable names indicate def-use relation.

Figure 4.6.: Intent correlation example in app Shanghai 1930.

rules. Line 11 is further correlated with line 8 because of variables `v8`, again by the *useUse*() rules. Hence, we have *correlatedIntent*(`PaySmsActivity.a()`, **SendSms**, 8, **SmsNotify**, 16)=TRUE. Intuitively, the two intents are correlated because the same content is being sent over a short message and written to the mail box. Thus, the message send is not stealthy.

4.3.2 UI Compatibility Check

After intents are propagated to top level functions, the next step is to check their compatibility with the text of the corresponding user interface artifacts.

Acquiring User Interface Text. Given a top level function, we need to first extract the corresponding text. User interface components in an Android app are organized in a view tree. A view is an object that renders the screen that the user can interact with. Views can be organized as a tree to reflect the layout of interface. There are two ways to construct the layout: (1) statically through an XML resource file; (2) dynamically by constructing the view tree at runtime.

With the static layout construction, upon the creation of an activity, the corresponding user interface is instantiated by associating the activity with the corresponding XML file by calling `setContentView([XML layout id])`. The Android core renders the interface accordingly. A UI object has a unique ID. The ID is often specified in the XML file. Inside the app code, the handle to a UI object is acquired by calling `findViewById([object id])`. For example, the following text defines a button in the XML file. Note that the button text is also specified.

```
<Button android:id="@+id/my_button"...
    android:text="@string/my_button_text"/>
```

Its handle can be acquired as follows. Note that the lookup id matches with that in the XML file.

```
Button btn = (Button)findViewById(R.id.my_button);
```

The event handler for an UI object is registered as a listener. For example, one can set the listener class for the previous button by making the following call.

```
btn.setOnClickListener(new MyListener(...));
```

In this case, the `onClick()` method of the `MyListener` class becomes the top level user interaction function associated with the button. Next we describe how we extract text for different kinds of functions.

For a top level *interactive* function *F* (e.g. `onClick()`), AsDroid identifies the corresponding UI text as follows. It first identifies the registration point of the listener class of *F*. From the point, AsDroid acquires the UI object handle, whose ID can be acquired by finding the corresponding `findViewById()` function. The ID is then used to scan the layout XML file to extract the corresponding text. AsDroid also extracts the text in the parent layout. For example, the parent layout of a button may be a dialog. Important information may be displayed in the dialog and the button may have only some simple text such as "`OK`". We currently cannot handle cases in which the text is dynamically generated. We found such cases are relatively rare.

Some *non-interactive* top level functions also have associated UIs, for instance, the lifecycle methods `onCreate()` and `onStart()` of activity components. These methods are invoked when the screen of an activity is first displayed. While no user interactions are allowed when executing these methods, the displayed screen may have enough information to indicate the expected behavior of these methods, such as loading data from a remote server. Hence, for an activity lifecycle method, AsDroid extracts the text in the XML layout file associated with the activity.

Text Analysis. Once we have the text, we build a dictionary that associates a type of intent to a set of keywords through training. We use half of the apps from the benign sources[2] as the training subjects, which account for about 28% of all the apps we study. During evaluation, we use the dictionary generated from the 28% apps to scan over the entire set of apps. Here, we assume the training apps are mostly benign. If an intent appears

[2]We collect apps from both benign and malicious sources as shown in Section 4.4.

Algorithm 3 Generating Keyword Cover Set.

```
train($S$, $F$)
1: $KWD=\phi$ /*the keyword cover set*/
2: while $F \neq \phi$ do
3:     sort $S$ by keyword (or keyword pair) frequency
4:     $k$=the top ranked keyword (or pair) in $S$
5:     $X$= the functions in which $k$ occurs
6:     $KWD=KWD \cup k$
7:     $F=F$-$X$
8:     $S$=$S$-{all the keywords (pairs) in $X$}
9: end while
```

together with some text in a benign case, then the intent and the text are compatible. We use keywords to represent text, and build compatible keyword cover set for each intent. In particular, For each intent type T of interest, we identify all the top level functions F that have T annotated and collect their corresponding texts. We then use Stanford Parser [84] to parse the text to keywords. We populate a universal set S to include all individual keywords and keyword pairs that appear in these functions. We then use Algorithm 3 to identify the smallest set of keywords (or pairs) that have the highest frequency and cover all the top level functions tagged with T.

The algorithm is similar to the greedy set cover algorithm [85]. It picks the most frequently occurring keyword k at a time and adds it to the keyword set. Then it removes all the keywords that appear in the top level functions in which k occurs, as they can be covered by k. It repeats until the set of functions are covered.

We consider keyword pairs are semantically more predictive. Hence, we first apply the algorithm to keyword pairs and keep the pairs that can uniquely cover at least 10% of functions. Then we apply the algorithm to singleton keywords on the remaining functions.

Figure 4.7 shows the generated keyword cover set for the **Send-Sms** intent. Observe some keywords are semantically related to the intent but some are not, e.g. “OK” and “Register”, which occur rarely but do uniquely cover some functions. Further inspection shows that it is due to the malwares in the training pool. Hence, we also use human semantic analysis to prune the keyword set, e.g. filtering out “OK” and “Register”. The keyword

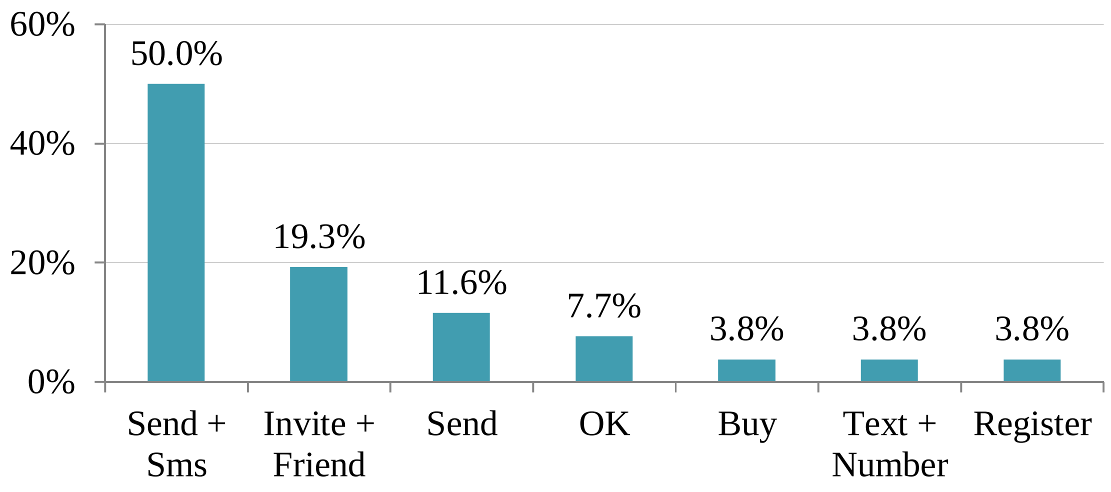


Figure 4.7.: The keyword cover set for the SendSms intent. The y-axis denotes the percentage of top level functions that can be *uniquely* covered by a keyword (pair).

set of **HttpAccess** is similarly constructed, containing keywords “Download”, “Login”, “Load”, “Register”, and so on. The cover set of **PhoneCall** is much simpler, containing only one keyword “Call”.

Once we get the keyword cover set, we further populate it with its synonyms, using Chinese WordNet [86] to have the final dictionary.

Compatibility Check. The compatibility check is performed as follows.

- Given a top level function F with UI text S and an intent T, if S is incompatible with T and all the intents correlated with T, it is considered a mismatch. Note that we consider empty text is incompatible with any intent.

- If T is a **SendSms** intent and has a correlated **SmsNotify** intent. It is not a mismatch regardless of the UI text.

- If T is **HttpAccess**, the technique checks if the corresponding UI text is compatible. If not, it further checks if T is correlated to any **UiOperation** intent. If not, the intent is consider stealthy. Intuitively, it suggests that even an HTTP access is not explicit from the GUI text, if the data acquired through the HTTP connection are used in

Table 4.1.: Experiment results.

		HTTP			SMS			CALL			INSTALL					
	#App	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)
Contagio	96	189(69)	136(64)	28/7(14/2)	90(57)	86(55)	0	4(4)	2(2)	0	4(2)	4(2)	0/7(0/6)	287(82)	228(77)	28/14(14/8)
Google Play	12	19(9)	12(7)	3/0(2/0)	6(5)	6(5)	2/0(1/0)	2(1)	0	0	0	0	0	27(10)	18(8)	5/0(3/0)
Wandoujia	74	166(39)	70(23)	23/5(10/1)	46(24)	13(10)	3/2(2/2)	8(5)	0	0	0	0	0	220(47)	83(28)	26/7(11/3)
Total	182	374(117)	218(94)	54/12(26/3)	142(86)	105(70)	5/2(3/2)	14(10)	2(2)	0	4(2)	4(2)	0/7(0/6)	534(139)	329(113)	59/21(28/11)

some UI component (e.g. fetching and then displaying advertisements from a remote server), the HTTP access is not considered stealthy.

4.4 Evaluation

We implement a prototype called AsDroid (*Anti-Stealth Droid*). We transform the DEX file of an app to a JAR file with dex2jar [87] and then use WALA [28] as the analysis engine. Our implementation is mainly on top of WALA.

We have collected apps from three different sources. We aim to detect those with the following stealthy behavior: SMS sends, phone calls, HTTP connections and component installations. Hence, we only focus on those having the permissions for such behaviors. Particularly, since almost all apps have the HTTP permission, we select those that have at least one of the other three permissions. Note that despite we introduce six intents in Section 4.3, **SmsNotify** and **UiOperation** do not describe stealthy behavior but rather suppress false alarms. The 3 sources are the following.

◇ **Contagio Mini Dump** [88]. It collects a large pool of (potential) malware reported by users and existing security tools. These malicious apps may perform stealthy operations, leak user private information, or compromise the operating system like a rootkit. We acquired 96 apps holding the needed permissions.

◇ **Google Play** [89]. This is the official apps market holding a lot of Android games. We checked the top 180 free game apps and only 12 of them satisfy our selection criteria.

◇ **Wandoujia** [90]. This is a popular general Android app market in China. We have checked the 1000 most popular game apps on the market and downloaded 74 of them with the needed permissions.

The detection results are shown in Table 4.1. In the table, #App in the second column denotes the number of tested apps from a specific source. #Intent is the number of API invocations with one of the four kinds of potential stealthy intents. #Rep is the number of intent points reported by AsDroid as stealthy. #FP is the number of false positives and #FN is the number of false negatives. The corresponding #App in parentheses denotes the number of apps in which these intents appear. Note that one app may have multiple intents. The last three columns show the total numbers. #App in the last three columns is not the simple sum of the #App in the corresponding preceding columns. For example, the number of total reported apps is 77 for the `Contagio` source. It is not the sum of the reported apps in the four categories as one app may be reported in multiple categories. We make the following observations.

- AsDroid is able to detect a lot of stealthy behaviors in these apps. Totally, AsDroid detects that 113 apps perform stealthy operations, with 85 true positives, i.e. having at least one true stealthy API call. Note that there are some apps that do not have the intents (i.e. API calls) of interest even though they hold the permissions. Since there are no existing oracles to determine stealthy behavior, we identify true positives by manually inspecting the results in two ways. For those API calls that can be reached by testing, we determine their stealthiness by executing the apps. Many of the API calls are difficult to reach without a complex sequence of user actions. Since we lack automatic test generation support, we perform code inspection instead. AsDroid detects a lot of stealthy behavior in the apps from `Contagio`, which is supposed to be a source hosting (highly likely) malwares. Most of the detected stealthy SMS sends and phone calls may cause unexpected charges. Most of the stealthy HTTP accesses are to notify the remote servers the status of device or the app (e.g. a mobile device becomes online). Some of them also leak critical user information.

- AsDroid produces some false positives (28 out of the 113 reported apps). They are induced by the following reasons: (1) AsDroid cannot analyze dynamically generated text associated with a UI component; (2) The dictionary we use is incomplete; (3) Some reported intents are along infeasible paths but AsDroid does not reason about path feasibility. The detection outcome for individual apps is denoted by the symbols on top of the bars and their colors in Figure 4.9. Also observe that most false positives belong to the category of HTTP accesses. Some of them are due to the incompleteness of our keyword dictionary. However most of them are essentially HTTP accesses in advertisement libraries. These accesses often download advertisement materials and store them to *external files* that are later read and displayed. Ideally, they are not stealthy as the materials are displayed. However AsDroid currently cannot reason about correlations through external resources, leading to false positives. Note that most existing static data flow analysis engines on Android have the same limitation. It should be easy to have an additional post-processing phase to suppress warnings from advertisement libraries.

- The number of false negatives is small (11 apps total). We manually inspect the apps that are not reported by AsDroid to determine false negatives. In particular, we use WALA to report all the API calls of interest and then we inspect them one by one manually. There are $182-113=69$ such apps. We found that AsDroid missed 11 malicious apps. Most of them are in the category of stealthy install. As such, the detection rate of AsDroid is $85/(85+11)=88\%$. The main reason for false negatives is that the current implementation cannot model some of the implicit call edges. There are also cases that native libraries are used to perform stealthy behavior, which is not handled by AsDroid. The false negative HTTP accesses mainly result from the in-accuracy of the text analysis. While AsDroid extracted keywords such as "`download`" and "`login`" that make the (stealthy) HTTP accesses compatible and thus not being reported, these accesses doesn't match the textual semantics.

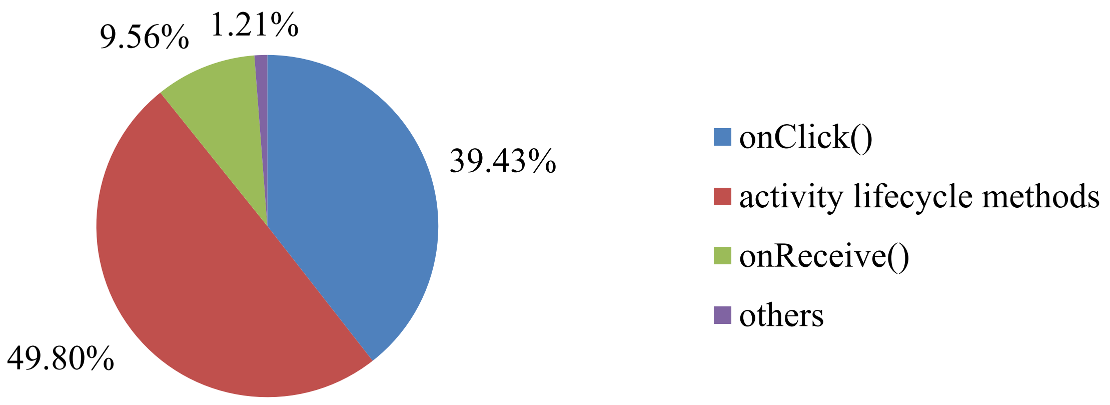


Figure 4.8.: Breakdown of the top level functions with intents. Activity lifecycle methods include `onCreate()` and `onStart()` of an activity. `onReceive()` and the other categories do not have associated UI.

- Stealthy HTTP connections are very common, although many of them may not be as harmful as the other stealthy behaviors (please refer to our case study). SMS sends are another dominant category of stealthy behaviors, which echoes the recent studies [19, 76].

Comparison with FlowDroid. FlowDroid [35, 36] is a state-of-the-art open-source static taint analysis for Android apps. We ran it on the 96 apps from `Contagio`. We use the default taint sources (e.g. methods retrieving private information). For the taint sinks, we only keep the SMS send and HTTP access methods. FlowDroid ran out of memory for 55 of the apps hence we compare the results for the remaining 41. FlowDroid reports 4 SMS sends in 3 apps and 1 HTTP access in 1 app that have information leak. In contrast, in the 41 apps, AsDroid reports 26 stealthy HTTP connections in 18 apps, including the one reported by FlowDroid, with 1 false positive in 1 app and 7 false negatives in 2 apps. It also reports 35 SMS sends in 21 apps, including 2 SMS sends reported by FlowDroid. For the other 2 SMS sends (by FlowDroid), the UIs explicitly indicate the behavior. Hence they are not stealthy although they do leak information. From the comparison, we clearly see that FlowDroid and AsDroid focus on problems with different natures. All experiments are performed on an Intel Core i7 3.4GHz machine with 12GB memory. The OS is Ubuntu 12.04.

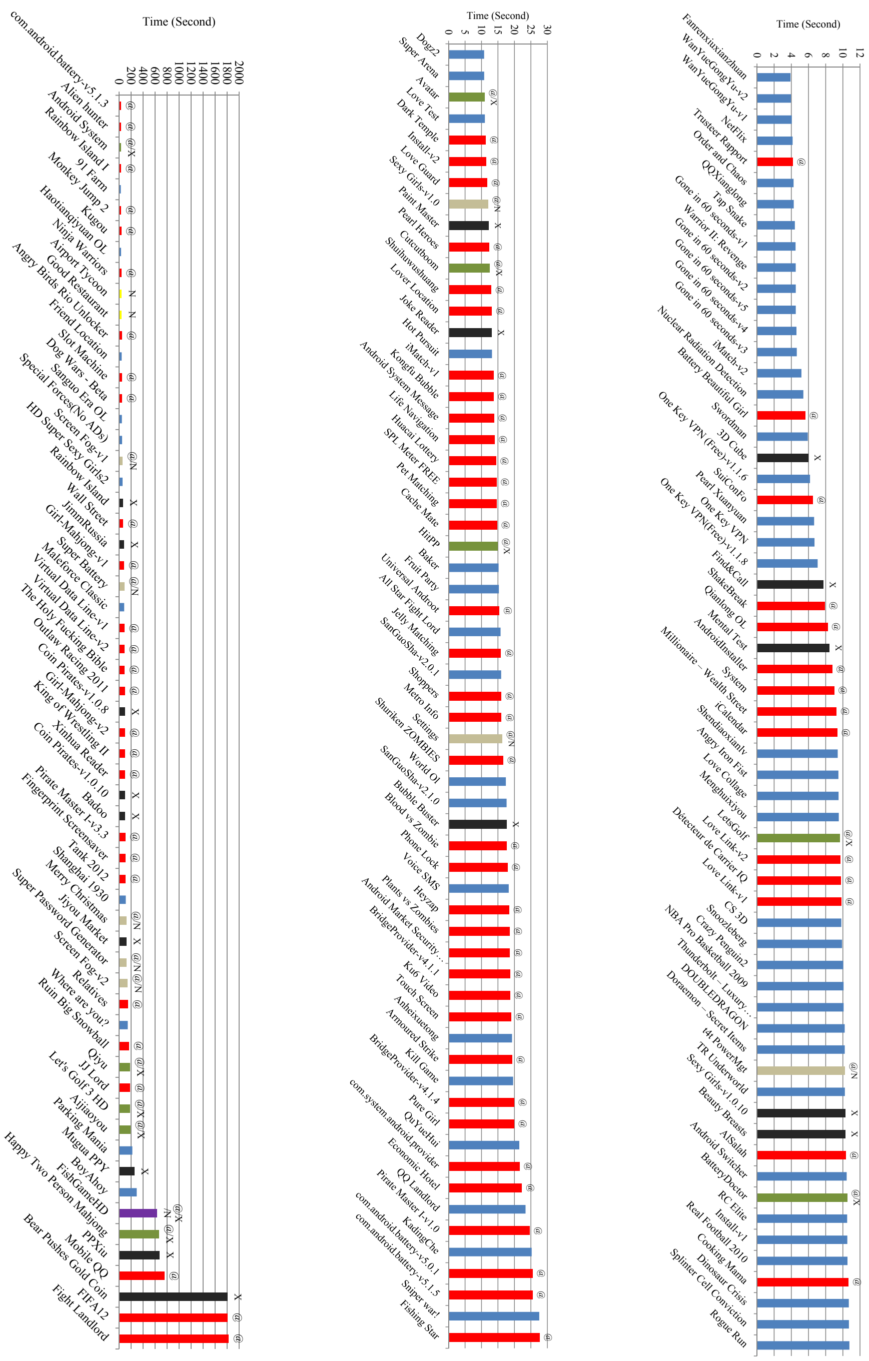


Figure 4.9.: Analysis time. The detection results are also annotated on top of each bar with '@' denoting true positive(*red*), 'X' false positive(*black*) and 'N' false negative(*yellow*). Since an app may have multiple intents, it may be annotated with multiple labels. The last 3 apps exceeded the max timeout 30 mins.

Figure 4.8 shows the breakdown of the top level functions that are attributed with intents. There are totally 743 such functions. Observe that 39% of such functions are the interactive `onClick()` function and almost 50% of them are activity lifecycle methods that are not interactive but nonetheless have associated UI. About 10% of them are `onReceive()` of external events and 1.2% of other functions such as the timer handler function `TimerTask.run()`. These functions are often not associated with any UI.

We present the analysis time for the 182 apps in Figure 4.9. Most apps (about 93%) can be detected in 3 mins and a few in 13 mins. Three apps require more than 30 mins. Human inspection disclosed that that they are very complex apps such that AsDroid consumes exceptionally large amount of memory, which slows down the analysis significantly. We plan to further look into this issue.

4.4.1 Case Studies

Next, we present two more cases.

iCalendar is a calendar app infected by malicious code that sends a SMS message subscribing to a premium-rate service. The malicious operation is triggered by user interaction in a stealthy way. The user clicks the app to change a background image and the app increases a counter. When the counter gets to 5, a message is sent. Figure 4.10 shows a simplified code snippet of the process.

Variable `main` represents the main interface layout. As soon as the app is launched, it registers a click listener in `onCreate()`. When the user clicks the interface, `showImg()` is invoked in `onClick()` to reset the background image. In the mean time, the app checks the counter to see if `sendSms()` should be called to send a premium-rate SMS.

In our analysis, two intents: **UiOperation** and **SendSms**, are associated with $\boxed{L_1}$ and $\boxed{L_2}$ in Figure 4.10 respectively. The intents are propagated to the top level function `onClick()` through the call graph. The UI component associated with the function is the background image without any text, which does not imply the **SendSms** indent. The correlation analysis also determines that these two intents are not correlated. It is hence reported as a

```
// in class {iCalendar}
public void onCreate(Bundle bundle) {
  main.setOnClickListener(this);
}
public void onClick(View view) {
  showImg();
}
private void showImg() {
  if(index == 5) {
    sendSms();
  }
  main.setBackgroundDrawable(drawable1); L1
}
public void sendSms() {
  smsmanager.sendTextMessage("106xxxx", null, "921X1", null,
      null); L2
}
```

Figure 4.10.: Code example in app iCalendar.

mismatch. Note that taint analysis tools [1, 35, 36] cannot report the problem because the data involved in the SMS send are hardcoded.

HitPP is a game app downloaded from Google Play. Figure 4.11 shows the code snippet in which a stealthy HTTP access is made when the app is initialized. The initialization at line 4 transitively starts a thread at line 14. The thread entry is at line 18. The thread starts an HTTP connection at line 21 and then shuts it off right after at line 22. The app does not receive or display any data from the remote server. We suspect the HTTP access is to inform the remote server about the start of the app. Since there is no UI text associated with the top level `onCreate()` method and there are no correlated intents, the HTTP access

```
 1 public class HitPP extends Activity {
 2   public void onCreate(Bundle bundle) {
 3     // {initialization} ...
 4     WiGame.init(this, "f11947a...", "Df6mBy...", true, true);
 5   }
 6 }
 7 class WiGame {
 8   public static void init(Context ctx, String s1, String s2,
         boolean x, boolean y) {
 9     b.a(ctx,s1);
10   }
11 }
12 class b {
13   public static void a(Context ctx, String str) {
14     (new b$1(str, ctx)).start();//→b$1.run() at line 18
15  }
16 }
17 class b$1 extends Thread {
18   public void run() {
19     String str = "http://d.wiXXX.com/was/r?u=" +
         WiGame.getDeviceId();
20     HttpGet httpGet = new HttpGet(str); //HttpAccess
21     httpClient.execute(httpGet); //without a LHS variable
22     httpClient.getConnectionManager().shutdown();
23  }
24 }
```

Figure 4.11.: Code example in app HitPP.

is reported by AsDroid. This is a very typical kind of stealthy HTTP access reported by AsDroid.

4.5 Limitations

AsDroid has the following limitations. (1) The current UI analysis is simply based on textual keywords, which may be insufficient. It is possible that apps use images or obfuscated texts (e.g. text containing keyword "send" but having no relation with sending a message). AsDroid will have difficulty in catching the intention of the UI. We will study applying more advanced text analysis or image analysis. (2) Currently, to avoid false positives, AsDroid relies on certain rules in detecting intent correlation and avoids reporting some intents incompatible with UI if their correlated intents are compatible. This seems to be working fine given that Android malwares are still in their early stage. In the future, if an adversary has the prior knowledge of AsDroid, he could obfuscate a malicious app to induce bogus correlations to avoid being reported. We envision a more sophisticated program analysis component will be needed, which may leverage testing or symbolic analysis (e.g. use symbolic analysis to determine if two intents are truely correlated). (3) AsDroid currently cannot reason about correlations through external resources, leading to false positives. Note that most existing static data flow analysis engines on Android have the same limitation. It could be mitigated by modeling external accesses. (4) Currently, AsDroid does not support native code or reflection. (5) AsDroid misses some Inter-Component Communication correlations. We could leverage Epicc [91] to get better coverage in our future work.

4.6 Related Work

TaintDroid applies dynamic taint analysis to Android apps [1] to prevent information leak. Gilbert et al. extended the technique to track implicit flows [92]. Hornyack et al. developed AppFench to impose privacy control on Android applications [38]. Arzt et al. investigated the limitations of using runtime monitoring for securing Android apps [93].

They used unintended SMS sending as an example. The essence of the technique is information flow tracking. FlowDroid [35, 36] is a very recent static taint analysis tool. These techniques cannot detect stealthy behavior as such operations may not leak information, as evidenced by the comparison with FlowDroid in Section 4.4.

Enck et al. developed a simple static analysis [2] that can detect SMS sends with hardcoded SMS numbers and phone calls, such as prefix "`tel:`" and substring "`900`". However, these patterns are very limited and not all such operations are malicious.

Elish et al. proposed to detect malicious Android apps [94] by determining the absence of data dependence path between user input/action and a sensitive function. However, dependence is not the key characteristic of stealthy behavior. In our experience, SMS sends triggered by user inputs can be malicious. Furthermore, many benign HTTP accesses are not triggered by any user action, e.g. an email app might connect to the server frequently to check new emails in background.

DroidRanger developed by Zhou et al. employs both static and dynamic techniques to detect malware [95], based on signatures derived from known malware such as premium-rate numbers and content of SMS messages. Hence, Droid-Ranger has to maintain a signature database that may change significantly overtime. And it also has runtime overhead.

Some existing work tries to capture Android GUI errors [96] or improve privacy control via GUI testing [97]. Gross et al. developed EXSYST [98] that uses search based testing to improve GUI testing coverage. Mirzaei et al. applied symbolic execution to generate test cases for Android apps [99, 100]. AsDroid could potentially leverage these techniques to generate test cases for bug report validation.

Recently, Pandita et al. proposed WHYPER to analyze an app's text description and then determine if the app should be granted certain permissions [7]. Both WHYPER and AsDroid leverage text analysis. However, they have different goals and AsDroid works by analyzing both apps and UIs.

4.7 Summary

We propose AsDroid, a technique to detect stealthy malicious behavior in Android apps. The key idea is to identify contradiction between program behavior and user interface text. We associate intents to a set of API's of interest. We then propagate these intents through call graphs and eventually attribute them to top level functions that usually have associated UIs. By checking the compatibility between the intents and the text of the UI artifacts, we can detect stealthy operations. We test AsDroid on 182 apps that are potentially problematic by looking at their permissions. AsDroid reports 113 apps that have stealthy behaviors, with 28 false positives and 11 false negatives.

5 CONCLUSION

This dissertation focuses on the topics of static analysis for Android apps. We first examined accurately discovering more sensitive data sources in Android apps. While existing researches focused on sensitive data disclosure detection with predefined data sources that are normally API functions, we proposed techniques to identify sensitive data sources among the generic API functions which may not generate sensitive data in many cases. Such generic API functions include reading data from user interface, files, network, etc. We leveraged text analysis to discover the sensitiveness. Then we applied bi-directional propagation to detect sensitive data disclosure issues. We also developed a technique to detect stealthy behaviors combining with text analysis and bi-directional propagation.

Detecting Sensitive User Inputs Disclosures. User inputs are very common in Android apps and many of them may contain sensitive information, *e.g.,* credit card number, birth date. Existing approaches of detecting sensitive data disclosures always focus on tracking the data generated by certain specific API functions. Such API functions directly returns sensitive data. The user input is a typical type of generic API functions that can return either sensitive data or insensitive data, depending on the context. However, existing work mostly neglect such data sources.

We develop SUPOR to detect the sensitiveness of the user inputs by inspecting the statically defined attributes and correlated text labels of the input fields. We mimic as real users to associate the input fields with correlated text labels and bind the discovered sensitive user input with corresponding API invocations in the code for further analysis. We evaluate our technique on a large number of real-world Android apps. The results show that it can effectively and efficiently identify sensitive user inputs and then detect their disclosures.

Detecting Even More Sensitive Data Disclosures. Besides the API functions for obtaining user inputs, there are more other generic API functions that can read data from files, network and other resources. Different with the user inputs that can be determined to be sensitive or not by examining the context of the user interfaces, we cannot easily determine the sensitiveness of those API functions by checking where they are used. If we ignore all such APIs, we may miss a lot of real problems when detecting sensitive data disclosures. But if we treat all such API functions as sensitive data sources, we can expect a lot of false warnings.

We develop type system based technique to decide whether a variable storing the data may hold sensitive information. We associate the variables with correlated text labels in the apps, either from the user interface or from the code. The text labels are treated as the types of the variables. Text analysis can then be applied to determine whether the associated text labels indicate the sensitiveness of the corresponding data. In case that the sensitiveness of a piece of data is discovered after the data falls into a sink point that discloses data to public channels, we allow the types to be propagated bi-directionally. While traditional techniques of sensitive data disclosure detection requires forward data flow paths from data sources to sinks, our technique is able to handle the cases in which backward paths exist from where we recognize the data sensitiveness to the sinks. We develop a prototype BIDTEXT and evaluate it on 10,000 Android apps. The results show the effectiveness and efficiency of our approach.

Detecting Stealthy Behaviors. Stealthy behaviors are the kind of behaviors that are executed without the users' consent. For example, malware may send a short message to a premium number in the background or make a phone call in the mid-night without any user actions. This kind of malicious behaviors cannot be easily distinguished from the benign ones because the benign ones perform the actions with the same API functions. Some existing techniques leverage the blacklist to identify malicious stealthy behaviors. For instance, if an SMS sending API function sends a message to a blacklisted number, it is reported as a stealthy behavior. But maintaining a blacklist is non-trivial and thus we need a more general approach to detect such behaviors.

We model stealthy behavior as the program behavior that mismatches with the user interface, which denotes the user expectation of program behavior. We assign an intent to each API function that indicates a specific behavior and propagate it backwardly to top level functions. A top level function may be a user interaction function with the behavior it performs. Then we extract the corresponding text from the user interface and examine whether the text information indicates the discovered program behaviors. Semantic mismatch of the two indicates stealthy behaviors. We develop AsDroid and evaluate it on a pool of Android apps that are potentially problematic. The results show that AsDroid is able to detect stealthy behaviors with low false positives and false negatives.

REFERENCES

REFERENCES

[1] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[2] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Conference on Security Symposium*, SEC'11, pages 315–330, Berkeley, CA, USA, 2011. USENIX Association.

[3] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 19th ACM SIGSAC Conference on Computer and Communications Security*, CCS'12, pages 229–240, New York, NY, USA, 2012. ACM.

[4] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in Android applications. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, NDSS'13, Reston, VA, USA, 2013. Internet Society.

[6] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, CODASPY'13, pages 209–220, New York, NY, USA, 2013. ACM.

[7] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22nd USENIX Conference on Security Symposium*, SEC'13, pages 527–542, Berkeley, CA, USA, 2013. USENIX Association.

[8] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. Autocog: Measuring the description-to-permission fidelity in Android applications. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security*, CCS'14, pages 1354–1365, New York, NY, USA, 2014. ACM.

[9] Kangjie Lu, Zhichun Li, Vasileios Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium*, NDSS'15, Reston, VA, USA, 2015. Internet Society.

[10] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE'03, pages 260–269, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics – Volume 1*, ACL'03, pages 423–430, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[12] The Stanford parser for natural language processing, 1999. `http://nlp.stanford.edu/`.

[13] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. The MIT Press, 1998.

[14] George A. Miller. Wordnet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, November 1995.

[15] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 1025–1035, New York, NY, USA, 2014. ACM.

[16] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[17] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting on Association for Computational Linguistics – Volume 1*, ACL'13, Stroudsburg, PA, USA, 2013. Association for Computational Linguistics.

[18] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: Scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys'12, pages 281–294, New York, NY, USA, 2012. ACM.

[19] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP'12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[20] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the 21st Annual Network & Distributed System Security Symposium*, NDSS'14, Reston, VA, USA, 2014. Internet Society.

[21] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the 9th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'88, pages 35–46, New York, NY, USA, 1988. ACM.

[22] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Notice*, 23(7):35–46, June 1988.

[23] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[24] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Notice*, 39(4):229–243, April 2004.

[25] Princeton University. Wordnet.
`http://wordnet.princeton.edu`.

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

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

[28] IBM T.J. Watson Research Center. Wala: T.J. Watson libraries for analysis.
`http://wala.sourceforge.net`.

[29] Eric H. Huang, Richard Socher, Christopher D. Manning, and Andrew Y. Ng. Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers – Volume 1*, ACL'12, pages 873–882, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics.

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

[31] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium*, NDSS'11, Reston, VA, USA, 2011. Internet Society.

[32] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, NDSS'12, Reston, VA, USA, 2012. Internet Society.

[33] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert Deng. Comparing mobile privacy protection through cross-platform applications. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, NDSS'13, Reston, VA, USA, 2013. Internet Society.

[34] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of the 20th ACM SIGSAC conference on Computer and communications security*, CCS'13, pages 1043–1054, New York, NY, USA, 2013. ACM.

[35] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 259–269, New York, NY, USA, 2014. ACM.

[36] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Notices*, 49(6):259–269, June 2014.

[37] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proceedings of the 21st Annual Network & Distributed System Security Symposium*, NDSS'14, Reston, VA, USA, 2014. Internet Society.

[38] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM SIGSAC Conference on Computer and Communications Security*, CCS'11, pages 639–652, New York, NY, USA, 2011. ACM.

[39] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*, CCS'13, pages 1029–1042, New York, NY, USA, 2013. ACM.

[40] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 1036–1046, New York, NY, USA, 2014. ACM.

[41] Jose Meseguer, Ralf Sasse, Helen J. Wang, and Yi-Min Wang. A systematic approach to uncover security flaws in GUI logic. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP'07, pages 71–85, Washington, DC, USA, 2007. IEEE Computer Society.

[42] Collin Mulliner, William Robertson, and Engin Kirda. Hidden gems: Automated discovery of access control vulnerabilities in graphical user interfaces. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP'14, pages 149–162, Washington, DC, USA, 2014. IEEE Computer Society.

[43] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 993–1008, Berkeley, CA, USA, 2015. USENIX Association.

[44] Christof Lutteroth. Automated reverse engineering of hard-coded GUI layouts. In *Proceedings of the 9th Conference on Australasian User Interface – Volume 76*, AUIC'08, pages 65–73, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[45] João Carlos Silva, Carlos Silva, Rui D. Gonçalo, João Saraiva, and José Creissac Campos. The GUISurfer tool: Towards a language independent approach to reverse engineering GUI code. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS'10, pages 181–186, New York, NY, USA, 2010. ACM.

[46] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE'10, pages 147–150, New York, NY, USA, 2010. ACM.

[47] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21(2):147–186, April 2014.

[48] OWASP. Information leakage.
`https://www.owasp.org/index.php/Information_Leakage`.

[49] MITRE. CWE-200: Information exposure.
`https://cwe.mitre.org/data/definitions/200.html`.

[50] FortiGuard Center. Information disclosure vulnerability in OpenSSL (Heartbleed).
`http://www.fortiguard.com/advisory/2014-04-08-information-disclosure-vulnerability-in-openssl`.

[51] US-CERT. OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160).
`https://www.us-cert.gov/ncas/alerts/TA14-098A`.

[52] Gartner. Gartner says smartphone sales surpassed one billion units in 2014.
`http://www.gartner.com/newsroom/id/2996817`.

[53] Search Engine Watch. Mobile now exceeds PC: The biggest shift since the Internet began.
`https://searchenginewatch.com/sew/opinion/2353616/mobile-now-exceeds-pc-the-biggest-shift-since-the-internet-began`.

[54] Collin Mulliner. Privacy leaks in mobile phone internet access. In *Proceedings of the 14th International Conference on Intelligence in Next Generation Networks*, ICIN'10, pages 1–6. IEEE, October 2010.

[55] Ning Xia, Han Hee Song, Yong Liao, Marios Iliofotou, Antonio Nucci, Zhi-Li Zhang, and Aleksandar Kuzmanovic. Mosaic: Quantifying privacy leakage in mobile networks. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM'13, pages 279–290, New York, NY, USA, 2013. ACM.

[56] Ning Xia, Han Hee Song, Yong Liao, Marios Iliofotou, Antonio Nucci, Zhi-Li Zhang, and Aleksandar Kuzmanovic. Mosaic: Quantifying privacy leakage in mobile networks. *ACM SIGCOMM Computer Communication Review*, 43(4):279–290, August 2013.

[57] Jinyung Kim, Yongho Yoon, and Kwangkeun Yi. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *Proceedings of 2012 Mobile Security Technologies*, MoST'12, 2012.

[58] Muhammad Haris, Hamed Haddadi, and Pan Hui. Privacy leakage in mobile computing: Tools, methods, and characteristics. *Computing Research Repository*, abs/1410.4978, 2014.

[59] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. Supor: Precise and scalable sensitive user input detection for Android apps. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 977–992, Berkeley, CA, USA, 2015. USENIX Association.

[60] Buycott.
`http://buycott.com/`.

[61] Michael Mahemoff. "Offline": What does it mean and why should I care?
`http://www.html5rocks.com/en/tutorials/offline/whats-offline/`.

[62] Nicholas C. Zakas. Towards more secure client-side data storage.
`https://www.nczonline.net/blog/2010/04/13/towards-more-secure-client-side-data-storage/`.

[63] Universal dependencies.
`http://universaldependencies.github.io/docs/`.

[64] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilhamy, Nguyen Nguyenz, and Martin Rinard. Information-flow analysis of Android applications in DroidSafe. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium*, NDSS'15, Reston, VA, USA, 2015. Internet Society.

[65] Wei Huang, Yao Dong, and Ana Milanova. Type-based taint analysis for Java web applications. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering – Volume 8411*, FASE'14, pages 140–154, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[66] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA'15, pages 106–117, New York, NY, USA, 2015. ACM.

[67] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security*, CCS'14, pages 1092–1104, New York, NY, USA, 2014. ACM.

[68] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 145–158, New York, NY, USA, 2007. ACM.

[69] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE'11, pages 11–20, New York, NY, USA, 2011. ACM.

[70] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tcomment: Testing Javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation*, ICST'12, pages 260–269, Washington, DC, USA, 2012. IEEE Computer Society.

[71] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 815–825, Piscataway, NJ, USA, 2012. IEEE Press.

[72] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. Automatic model generation from documentation for Java API functions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE'16, pages 380–391, New York, NY, USA, 2016. ACM.

[73] Gartner. Gartner says worldwide sales of mobile phones declined 3 percent in third quarter of 2012; smartphone sales increased 47 percent.
`http://www.gartner.com/it/page.jsp?id=2237315`.

[74] Juniper Networks. Juniper mobile security report 2011 – Unprecedented mobile threat growth.
`http://forums.juniper.net/t5/Security-Mobility-Now/Juniper-Mobile-Security-Report-2011-Unprecedented-Mobile-Threat/ba-p/129529`.

[75] TrendLabs. 3Q 2012 security roundup – Android under siege: Popularity comes at a price.
`http://www.trendmicro.com/us/security-intelligence/`.

[76] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'11, pages 3–14, New York, NY, USA, 2011. ACM.

[77] Denis Maslennikov. IT threat evolution: Q1 2013.
`http://www.securelist.com/en/analysis/204792292/`.

[78] Michael Becher, Felix C. Freiling, Johannes Hoffmann, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile security catching up? Revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP'11, pages 96–111, Washington, DC, USA, 2011. IEEE Computer Society.

[79] Paul Gosling. Trojans & spyware: An electronic achilles. *Network Security*, 2005(3):17–18, March 2005.

[80] Money-stealing apps are hosting in the mobile devices.
`http://finance.sina.com.cn/money/lczx/20120410/070311783396.shtml`.

[81] Google. Android 4.2 compatibility definition.
`http://source.android.com/compatibility/4.2/android-4.2-cdd.pdf`.

[82] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Pearson Education, Inc., 2006.

[83] Google. Android developer guide.
`http://developer.android.com/guide/`.

[84] Roger Levy and Christopher Manning. Is it harder to parse Chinese, or the Chinese treebank? In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics – Volume 1*, ACL'03, pages 439–446, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[85] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[86] National Taiwan University. Chinese wordnet.
`http://lope.linguistics.ntu.edu.tw/cwm/`.

[87] pxb1988. dex2jar: Tools to work with Android .dex and Java .class files.
`http://code.google.com/p/dex2jar/`.

[88] Contagio mobile malware mini dump.
`http://contagiominidump.blogspot.com/`.

[89] Google play market.
`https://play.google.com/store/apps/`.

[90] Wandoujia.
`http://www.wandoujia.com/apps/`.

[91] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Conference on Security Symposium*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[92] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated security validation of mobile apps at app markets. In *Proceedings of the Second International Workshop on Mobile Cloud Computing and Services*, MCS'11, pages 21–26, New York, NY, USA, 2011. ACM.

[93] Steven Arzt, Kevin Falzon, Andreas Follner, Siegfried Rasthofer, Eric Bodden, and Volker Stolz. How useful are existing monitoring languages for securing Android apps? In *ATPS*, volume P-215 of *GI Lecture Notes in Informatics*, pages 107–122. Gesellschaft für Informatik, 2013.

[94] Karim Elish, Danfeng (Daphne) Yao, and Barbara G. Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Proceedings of 2012 Mobile Security Technologies*, MoST'12, 2012.

[95] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, NDSS'12, Reston, VA, USA, 2012. Internet Society.

[96] Sai Zhang, Hao Lü, and Michael D. Ernst. Finding errors in multithreaded GUI applications. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA'12, pages 243–253, New York, NY, USA, 2012. ACM.

[97] Antti Jääskeläinen. *Design, Implementation and Use of a Test Model Library for GUI Testing of Smartphone Applications*. Doctoral dissertation, Tampere University of Technology, Tampere, Finland, January 2011.

[98] Florian Gross, Gordon Fraser, and Andreas Zeller. Exsyst: Search-based GUI testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1423–1426, Piscataway, NJ, USA, 2012. IEEE Press.

[99] Nariman Mirzaei, Sam Malek, and Riyadh Mahmood Corina S. Păsăreanu, Naeem Esfahani. Testing Android apps through symbolic execution. In *Proceedings of the 2012 Java Pathfinder Workshop*, JPF'12, New York, NY, USA, 2012. ACM.

[100] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, November 2012.

VITA

VITA

Jianjun Huang was born in Shehong, Sichuan Province, China, in 1986. He received the B.E. degree in information management and information system, and the M.S. degree in systems theory from the Renmin University of China in 2009 and 2012, respectively. He attended Purdue University from 2012 through 2017, studying program analysis with Professor Xiangyu Zhang. He received his Ph.D. degree in computer science in 2017. After graduation from Purdue University, he joined the Renmin University of China as an assistant professor.