PRIVSENSE: STATIC ANALYSIS FOR DETECTING PRIVACY-RELATED CODE IN MOBILE APPS

by

WAYNE HAVEY

B.S., University of Colorado Colorado Springs, 2017

A thesis submitted to the Graduate Faculty of the

University of Colorado Colorado Springs

in partial fulfillment of the

requirements for the degree of

Master of Engineering

Department of Computer Science

2020

This thesis for the Master of Engineering degree by

WAYNE HAVEY

has been approved for the

Department of Computer Science

by

Yanyan Zhuang, Chair

Terrance Boult

Sang-Yoon Chang

Date July 27th 2020

Havey, Wayne (M.E., Information Assurance).

PrivSense: Static Analysis for Detecting Privacy-Related Code in Mobile Apps

Thesis directed by Professor Yanyan Zhuang.

ABSTRACT

Android application data leakage remains pervasive, despite advancements in leakage detection and user protections. Previous research includes tools that are computationally expensive or that did not distinguish between the types of data that were leaked. For example, a user's age was considered as privacy-sensitive as their passwords.

This thesis introduces PrivSense, a light-weight tool designed to detect privacy-related code. PrivSense uses Natural Language Processing techniques to capture the semantic meanings of variables, classes, methods, etc., and determines the private data an app is likely to collect. Further, PrivSense correlates the sensitivity of private information with consumers' attitudes toward data sensitivity. We analyzed 925 apps across 35 categories, and found apps superfluously collect private consumer data ranging from a social security number and banking information, to medication, fingerprints, and physical location. Some apps are wrongly categorized by the Google Play Store which could cause consumer confusion. False positive and false negative rates of PrivSense are analyzed and presented.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

TABLE

LIST OF FIGURES

FIGURE

CHAPTER 1

INTRODUCTION

Mobile applications have been found to collect sensitive information from users reportedly in order to "utilize user information for better services" [1]. However, many mobile apps today collect information from users beyond what is needed for the apps to function, often without users' knowledge. Google, for example, uses data collected from the Android system as part of a $20 billion dollar location-based ad market. Law enforcement subpoenas this data for help in investigations [2]. Mining data from GPS signals, cellphone towers, nearby WiFi devices, and Bluetooth beacons can illustrate a pattern of life that regulatory bodies have not yet understood. Data collection in mobile apps needs to be investigated and aligned to public consensus of sensitivity.

A plethora of research has conducted static and dynamic analysis of mobile apps to understand their patterns of data collection, especially on the Android systems due to their popularity in the mobile market. However, most research tools involve complex models of the app lifecycle, data flow, and so on. This is mostly because the ecosystem of the Android OS is complex, and this makes it inherently difficult to analyze an app's behavior. First, unlike C or Java programs, no main function or method exists in an Android application. Many entry points are *implicitly* called by the Android system, e.g., upon pausing or resuming an app. The system can even stop an app because of low memory and later restart it. Second, the Android OS allows developers to register callbacks for various events, like the change of location

or UI interactions. None of the callbacks can execute in a pre-determined order. Therefore, building code paths, models, or function call graphs becomes challenging, thus causing existing tools on Android to be computationally expensive, like FlowDroid [3], SmartDroid [4], DroidChecker [5], Pegasus [6], and others [7–10].

In this paper, we design and implement a light-weight tool called PrivSense that analyzes Android apps at the source code level. Through the use of Natural Language Processing (NLP) techniques, it is not affected by the asynchronous lifecycle of Android apps. Our intuition is that any identifiers in code, like variables, classes, methods, are named with some semantic meaning, e.g., gender, name, and so on [11]. By extracting identifiers and analyzing their semantic similarity with sensitive keywords, PrivSense can detect privacy-related identifiers (we name them *privacy identifiers* or *identifiers* hereafter) that are likely relevant to an app's behavior that collects user's private information. Different from prior work, including those using NLP techniques [11], PrivSense is both light-weight and easy to prototype.

Furthermore, PrivSense ranks the *sensitivity level* of its detected private identifiers, based on real consumer's perceptions about how sensitive their personal data is [12, 13]. For example, most people think that financial account numbers and passwords are more sensitive compared to their occupation [13]. By incorporating a *privacy score*, each identifier is given a weight that measures the sensitivity level of the data collected by an app. We present the results of PrivSense giving a score to all identifiers extracted from an app, and compare apps across various categories. To the best of our knowledge, we are the first to incorporate consumer's privacy perceptions in the analysis of Android apps.

Our contributions can be summarized as follows:

- **A light-weight tool for detecting privacy-related code**. PrivSense uses NLP techniques to capture the semantic meanings of variables, classes, methods, etc. Its analysis is light-weight, and not affected by the asynchronous lifecycle of Android apps.

- **A consumer-centered privacy scoring mechanism**. PrivSense correlates the sensitivity of private information with the U.S. consumers' attitudes toward data sensitivity by providing a privacy score to each identifier detected.

- **Identification of alarming data collection behavior**. We analyzed 925 of the most popular apps across 35 app categories from the Google Play Store to determine how privacy-related keywords are used by developers. We found that apps superfluously collect consumer data from social security numbers and banking information to medication, fingerprints, and location. Some apps are wrongly categorized, which could cause consumer confusion.

- **False Positive and False Negative Analysis**. Unbiased research assistants that are unfamiliar with the code were enlisted to manually analyze the results for several Apps to help determine the prevalence of false positives. In order to analyze the possibility of false negatives, the thresholds for considering an identifier to be an instance of the app collecting privacy sensitive information were reduced and the results were analyzed.

The rest of the paper is organized as follows. Section 2 provides an overview of the relevant research. Next, in Section 3 we present the design and implementation

of PrivSense. In Section 4 we evaluate PrivSense by analyzing 925 popular apps, and then we discuss PrivSense's limitation in Section 5 and dicuss its false positive and false negative results. Finally, Section 6 concludes.

CHAPTER 2

RELATED WORK

As the Android ecosystem has the largest market share of mobile devices, it has also attracted a diverse set of available applications. Benign applications coexist with applications that do not respect privacy policies, or violate them unknowingly [14]. It can be difficult to determine developer intent thus motivating researchers to evaluate the app for suspicious behaviors instead [15]. Detecting malicious behavior within applications occurs primarily in three ways. Static and dynamic analysis, or a fusion of the two [16], are used to determine common libraries, function names, variables, as well as to investigate the requested application permissions, network communications, and UI interactions [10, 16]. Other directions to secure Android applications include modifying the base OS or repackaging the application to include code that protects the host [17]. Finally, virtualization is used to sandbox applications to isolate applications running on the same device from each other [18].

To address the gap between the privacy policies the application discloses, permissions it requests, and what actions the application actually takes, researchers began to evaluate apps for privacy sensitive APIs and then generating corresponding privacy policies programmatically [19]. Application scan speed and accuracy improved upon prior work [10] by addressing word ambiguity, that is the coexistence of multiple meanings for a word or phrase. More recently, NLP techniques have been applied to static analysis on Android, through finding the semantic meaning of code [11, 20]. Our work differs from prior research in that our NLP solution is light-weight, takes

into account users perceptions of privacy, and analyzes non standard identifier names found in source code.

Current NLP tools do not differentiate the "sensitivity level" of code. For example, is a person's job title more sensitive from a privacy perspective than their date of birth? Research has shown that not all sensitive data is created equal. Societal attitudes regarding sensitive data differ depending on what the data is related to. Markos, et al [12] surveyed US and Brazilian attitudes regarding privacy-related data and found that data like an individual's occupation, their political affiliation, and race were considered less sensitive than other data like financial account numbers, passwords, or characteristics that could not be changed, like fingerprints. Schomakers, et al [13] later expanded this work to include German attitudes towards data sensitivity. US, Brazilian, and German attitudes towards this ranking of different sensitive data fields largely trend in the same direction with few exceptions. It is thought that education levels across the survey data are the cause of the gap between nations [12]. PrivSense differs from previous tools in that it bases its ranking of the sensitivity level of keywords on real consumer perceptions. Therefore, it can create a more accurate profile of an app's behavior that more aligns with public attitudes. So far, such a privacy scoring mechanism has only been proposed in the context of social networks [21].

CHAPTER 3

DESIGN

PrivSense first downloads an app's APK from the Google Play Store before decompiling the code. The code is then fed into the *Identifier Extraction* module that extracts identifiers such as variable or method names. Following this, PrivSense uses the *Similarity Analysis* module to detect keywords that are privacy related amongst these extracted identifiers. Specifically, it uses a list of predetermined privacy keywords and weighs these keywords in a manner that corresponds to consumer sentiment. These steps are outlined in Figure 3.1. The following section describes each of these steps in further detail.

3.1 Identifier Extraction

From a decompiled Android apps source code, each Java based file is a compilation unit that contains syntactic elements such as classes, methods, attributes, and so on. Most of these syntactic elements have unique identifiers, or names specified by the

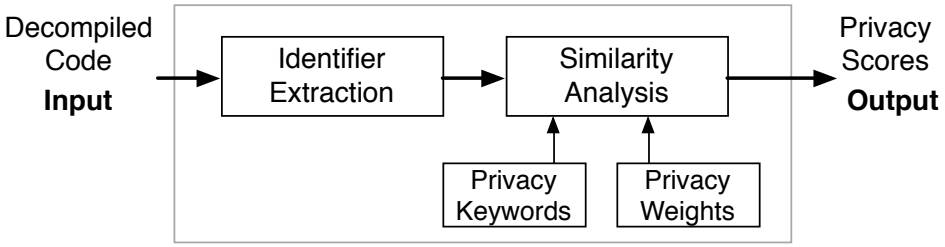

Figure 3.1: Overview of PrivSense.

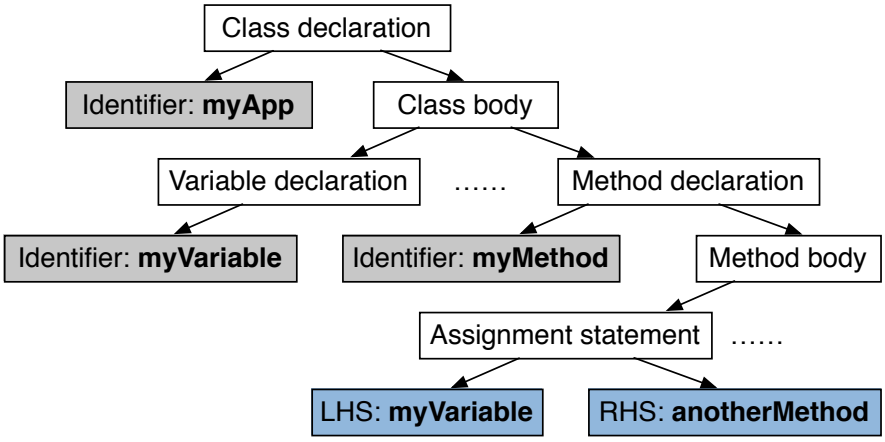

Figure 3.2: An example identifier tree for a java class named `myApp`. Identifier names are in bold. Gray nodes indicate identifier creation, and blue nodes indicate identifier reference. White nodes are intermediate steps that lead to identifier creation or reference. Dots $(\cdots)$ mean that multiple syntactic elements could exist at the same level.

developer. The intuition behind PrivSense is that identifiers usually carry semantic meanings, such as age and gender. Once the identifiers are extracted, they can be compared against a set of privacy related keywords to determine the likelihood of an application collecting private information.

To efficiently extract identifiers from a large number of source code files, we built a tree-like data structure, called an *identifier tree*, for each compilation unit. Each identifier tree comprises of a root node that is typically the top-level class in the Java source, with child nodes that represent the nested syntactic elements within the same class. To build such a tree, we traverse each compilation unit from its root. In each step of the traversal, if a new identifier of a class, method, variable, etc. is found, we extracted it as a child node, and appended the node to its parent. When multiple instances of the same identifier are used in the same Java source, we increment a

count stored in the node. An example identifier tree is shown in Figure 3.2 for a Java class called `myApp`. Identifier names are in bold, e.g., `myVariable`, `myMethod`. Gray nodes indicate identifier creation, and blue nodes indicate identifier reference. White nodes are intermediate steps that lead to either identifier creation or reference. Finally, all the identifier trees are used as the input to the Similarity Analysis module.

3.2 Similarity Analysis

This module determines similarity between extracted identifiers from the Identifier Extraction module, and a predefined set of privacy related keyword based on Levenshtein [22] distance and semantics similarity [23].

Levenshtein similarity is a way to measure the structural differences between words. It calculates the Levenshtein distance by measuring the number of changes, such as deletions or substitutions, needed to transform word `w1` to word `w2`, when `w1` and `w2` are the subjects of comparison. Due to the nature of programming languages, programmers often choose identifiers with special characters like underscores (`my_medication`), numbers (`medication1`), or with multiple words combined in a single identifier (`myMedication`). However, for readability, programmers often choose identifiers with derivable meanings. Therefore, identifiers are usually some variant of meaningful words. Since the majority of identifiers are unlikely to be part of the English vocabulary, it is necessary to compare them to the predefined, privacy-related keywords through measuring their structural differences. In the examples above, none of `my_medication`, `medication1`, or `myMedication` was found to be similar

Table 3.1: Privacy-related keywords and privacy scores.†

Privacy score	Keywords
5	citizenship, demographic_info, gender, job_title, marital_status, native_language, weight, height, political_affiliation, hair_color
6	birth_place, date_of_birth, age, religion, sexual_preference
7	username, address, location, income, currency, vehicle_registration, salary, email, phone_number, social_media_url, ssid
8	maiden_name, serial_number, credit_score, device_id, ip_address, mac_address, vehicle_vin
9	medication, digital_signature, drivers_license, fingerprint, insurance_policy_number, passport_number
10	password, bank_account_number, credit_card_number, house_financial_info, social_security_number

†Our list of privacy-related keywords was obtained through surveying the current literature on Android privacy research. The work of Markos et al. [12] and Schomaker et al. [13] contained a broader set of keywords. Table 3.1 shows our list mapped to the scale of Markos et al. [12] and Schomaker et al. [13]. Our list does not contain keywords with a score between 1 through 4 in their scale.

to the word "medication" using a semantic model for the English language. However, they can be compared against the keyword "medication" through Levenshtein similarity with accurate results.

Semantic similarity is a metric defined over a set of documents or terms, by grouping together terms that are closely related and spacing apart the ones that are distantly related [23]. It is a direct comparison of words where the word structure is irrelevant. It is necessary to use semantic similarity to augment the Levenshtein comparison. For example, words "vehicle" and "car" have a similar semantic meaning, but their Levenshtein similarity indicates they are very different.

The Similarity Analysis module parses the identifier trees generated by the Identifier Extraction module, by first removing identifiers that are known to be part of the Android ecosystem, or common enough to not warrant checking. Some examples include `onCreate`, `button`, `view`, and so on. Next, the Similarity Analysis

module compares each of the remaining identifiers against a pre-determined list of privacy-related keywords. We compiled a list of such keywords by surveying the current literature on Android privacy research (suc as [7–10]). The results are shown in Table 3.1. To determine similarity, thresholds are defined for both Levenshtein and semantic similarity. We used several combinations of thresholds until we found a good medium erring on the side of less false positives (strategies to reduce false positives and negatives are discussed in Section 5). If an identifier's similarity score exceeds either threshold, the identifier and some metadata (e.g., the keyword this identifier is similar to) is added to the results. These results are used in the next step.

3.3 Privacy-Related Keywords and Scoring

Not all keywords in Table 3.1 are equally sensitive. For example, most people think that financial information is more sensitive compared to their emails [13]. To consider this, we use a privacy score to measure the sensitive level of each keyword in Table 3.1. Leveraging the scale provided by Markos et al. [12] and Schomaker et al. [13], we weighted the keywords to correlate with their survey results from the United States, with a score of 1 being the least privacy sensitive, and 10 being the most sensitive. Fractional values were rounded up to the nearest whole number. The scores of all the keywords are listed in the first column of Table 3.1. According to the Similarity Analysis module, if an identifier `i` is similar to a keyword `k` that has a privacy score of `s`, then identifier `i`'s privacy score is also `s`. From these scores, we

can measure how sensitive the information an app is likely to collect. This will be evaluated in Section 4.

CHAPTER 4

EVALUATION

We evaluated PrivSense in the following ways, after analyzing a total of 925 apps across 35 categories. We first examined the overall privacy related behavior of all categories (Section 4.1). Next, we looked into several cases where an app's behavior deviates from apps in the same category (Section 4.2). Analysis of false positives and false negatives are presented in Section 4.3.

4.1 Overall Results

Our results show that 717 out of 925 apps (78%) had at least one private identifier and a total of 31,328 unique identifiers were found. This means that among the apps we analyzed, an average of 34 unique private identifiers were used in each app. However, we believe that this only a lower bound (reasons discussed in Section 5).

Among the 35 app categories, the category with the most identifiers found is 'Shopping' with 2,635 total unique findings. The category with the highest average privacy sensitivity score is 'Food and Drink' with an average score of 7.43 out of 10. Figure 4.1 shows the average score in each app category. Some categories have a score lower than 5, because no privacy identifiers were found in some apps. Even so, 28 out of 35, or 80% of the categories have an average score between 5 and 8.

To understand which identifiers contributed to a high average score, we analyzed the app categories with the top 10 highest scores in Figure 4.1 and the identifiers

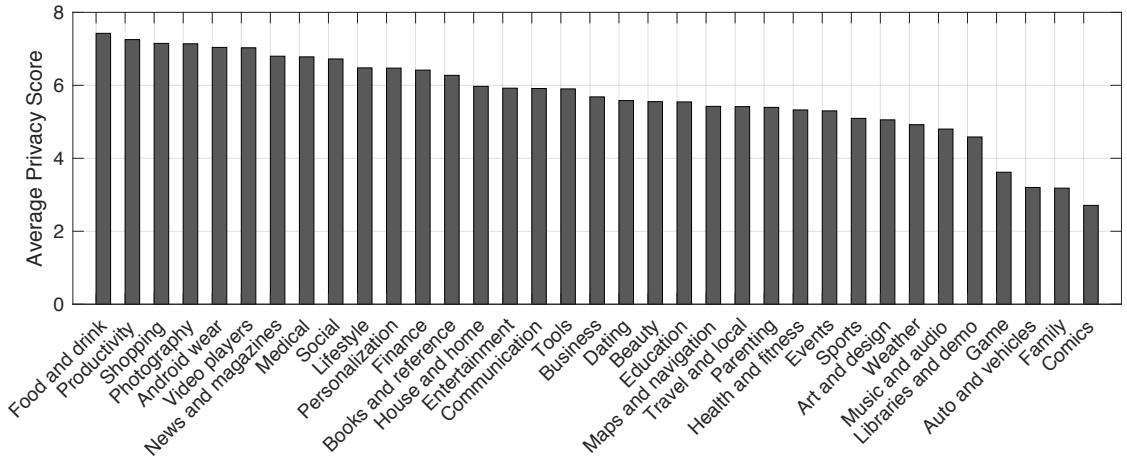

Figure 4.1: Average privacy score in each category. Some categories have an average score lower than 5 because we found no identifiers from some apps in those categories that are similar to the keywords in Table 3.1.

with a score of 10 in each of these categories. Most of these app categories collect a credit card number, particularly 'Food and Drink', 'Shopping', 'Medical', and 'Social'. One expects that food- or shopping-related apps would require a credit card number to make in-app purchases. However, many of these apps should not have any purchasing functions. For example, an app called Instant Pot belongs to the 'Food and Drink' category, but it is an app for recipes [24], and it still collects credit card information. Another 'Social' app Taimi [25] also collects a credit card number, while being a LGBTQI+ dating, chat, and social platform. Furthermore, many apps in the 'Medical' and 'Shopping' categories also collect a social security number in addition to a credit card number.

4.2 App Outliers

During our analysis, we also discovered apps whose behavior deviated from other apps in the same category. Table 4.1 shows some of these apps. Below is a summary of our findings.

4.2.1 Superfluous Data Collection

We define superfluous data as the private information collected but not intended for an app's function. We discovered such data ranging from personally identifiable information like a social security number (with a privacy score of 10), to medication and biometrics like fingerprints (with a privacy score of 9).

Personally identifiable information. As shown in Table 4.1, AliExpress, a 'Shopping' app based in China that offers retail services to international online buyers, collects a social security number and passport number. While seemingly superfluous, tech companies in China have to comply with the government's intelligence gathering operations [26]. For example, foreigners in China have been asked to submit their ID or passport information with their pictures from their devices, before they can use any in-app payment service [27]. The U.S. version of AliExpress, with over 9.2 million downloads, seems to extend the surveillance capability to users outside China, while their privacy policy does not disclose this [28]. As foreign apps accumulate influence in the U.S. [29], concerns on consumer privacy protection also grow significantly.

Medication and biometrics. Other apps that collect data superfluously include

Table 4.1: Apps with privacy scores that deviate the most from their category average.

App category	**App name**	**Score diff.**	**Keyword (Privacy score)**
Shopping	AliExpress	0.66	address (7), passport_number (9), credit_card_number (10), social_security_number (10)
Lifestyle	Nest	1.41	ssid (7), medication (9)
Parenting	Baby Tracker	2.82	date_of_birth (6), device_id (8), medication (9)
Communication	BOSS Revolution	2.36	phone_number (7), medication (9), credit_card_number (10)
Tools	My Spectrum	2.24	address (7), mac_address (8), ip_address (8), fingerprint (9)
Health and Fitness	Replika	2.82	income (7), phone_number (7), email (7), fingerprint (9), password (10)
Libraries and Demo	V380s	3.79	email (7), location (7), ip_address (8), device_id (8)
Sports	The Athletic	2.69	address (7), social_media_url (7), currency (7), location (7)
News and Magazines	Neighbors by Ring	1.17	address (7), location (7), phone_number (7)

Nest, a thermostat control app that also collects medication; Baby Tracker (a 'Parenting' app) and BOSS Revolution (an app for making international calls) both collect medication information; and My Spectrum, a tool for troubleshooting WiFi networks, also collects the user's fingerprint. Several other similar apps are listed in Table 4.1.

4.2.2 App in a Wrong Category

During our analysis, we discovered Neighbors by Ring, an app in the 'News and Magazines' category, collects both address and location. Location differs from address in that location is collected through GPS, cell towers, WiFi networks, etc. that provides finer grained information than address. Being in the 'News and Magazines' category, this seems odd. By reading its description, we discovered that Neighbors by Ring is a real-time crime and safety alert app. Therefore, location data is critical for it to function. When an app belongs to a wrong category, this can cause consumer confusion as what the app could do.

These results show that apps often over collect privacy-sensitive data than necessary, or contain wrong information in the Play Store. More in-depths analysis is needed to understand how the over-collected data is used.

4.3 Result Analysis

4.3.1 False Positive Analysis

PrivSense uses semantic similarity and Levenshtein thresholds for determining

whether an identifier is similar enough to one of the predefined privacy related keywords. These thresholds are configurable but one threshold was used for each similarity algorithm (Semantic and Levenshtein) in our initial analysis of applications from the Google Play store. To check for false positives, the help of three unbiased university students was enlisted to manually review the results and the associated app source code. These students were asked to review a reported identifier's context in the source code, and make a determination as to whether or not PrivSense's finding is indeed indicative of the collection of privacy related information. Additionally, the students were asked to record:

1. Their own subjective sensitivity rating.

2. How difficult it was to make their determination on scale of ***easy****: not many additional files or sections of code had to be reviewed,* ***medium****: some context had to be reviewed from surrounding code or other source code files,* ***hard****: many other source code files had to be reviewed.*

3. Whether or not the code surrounding the identifier appeared to be intentionally obfuscated.

4. Whether or not they would expect to find the identifier or the use of its associated sensitive content in the app based on what the app is used for. An example is discussed later in this section.

5. Any additional notes they found relevant.

There were a total of 399 identifiers reviewed by the students. Some of these are

duplicates for when the same identifier was used as multiple syntactic element types (same identifier used for a function name and variable name, for example). Of those, 59 unique identifiers were deemed false positives, which gives a false positive rate of 14%. This may seem to be a high percentage of false positives. However, there are many caveats that need to be addressed to understand this number.

First of all, the students are subjective reviewers and their determination of whether or not an identifier is privacy sensitive is based on their own opinion. The students' opinions may or may not reflect the opinions of the majority. As an example, one student determined that the identifier `company_name` was not privacy sensitive and concluded it was a false positive, even though the student noted that the identifier was used to save the user's company name to a profile. Many users especially in the government and military sectors would likely find this to be privacy sensitive. Additionally, the students may not have reviewed the source code thoroughly enough to make an accurate determination. The students were asked to record the specific source code files reviewed. Several false positives for one set of student results were marked as a false positive after only reviewing the R.java file. The R.java file is an auto-generated file used as an index for components defined in the main activity of the app. It indicates that the actual code that correlates with an identifier needs to be reviewed elsewhere. Several identifiers were marked by the students as false positives when they thought it was not actually related to the predefined keyword even if they noted that the identifier itself did in fact correlate to the collection of privacy sensitive data. For example, a Google library is used for collecting the users geo-location which is clearly privacy sensitive. However, this was

marked as a false positive since the identifier *Google* was not actually related to the predefined keyword of *Twitter*.

In the above, the reasons why the rate of false positives are likely lower than reported were addressed. However, there were several cases of real false positives. Many examples have to do with when an identifier reported would in fact correlate to the collection of privacy sensitive data, but when reviewed, it was in relation to different entity than the user of the application. For example, in the *Ancestry* app the identifier of `birth_date` was reported, which would be privacy sensitive if it was collecting the birth date of the user, but it was used for birth dates of ancestors.[1] Besides identifiers related to entities other than the user, there were still many false positives of identifiers that are commonly used in the Android ecosystem despite the presence of a white list (discussed in Section 5). One example is the identifier `message`, which has a 61% semantic similarity to the predefined privacy sensitive keyword of `email`. The identifier `message` made up 38% or 47 of the false positives before removing duplicates across all students evaluations. *Message* relates to a built in Android Class *Message* class and is often found as an identifier in relation to a commonly used function called *handleMessage* which takes *message* as an argument. Adding more identifiers to the white list of PrivSense is an easy way to reduce the false positive rate.

As previously mentioned, the students were asked to record additional information besides whether or not a reported identifier was a false positive. The following list summarizes this information:

[1]Some readers may argue that this too should be considered privacy sensitive especially if the *ancestor* is only one or two generations older than the user.

1. Difficulty to determine false positive status:
    (a) Easy: 285
    (b) Medium: 75
    (c) Hard: 36
2. Whether or not the privacy sensitive data related to the identifier was expected to be found in the application:
    (a) Expected: 308
    (b) Not Expected: 86
3. Whether or not the identifier and surrounding code appeared to be intentionally obfuscated:
    (a) Obfuscated: 15
    (b) Not Obfuscated: 276

From **1** an insight that can be gleaned is: **PrivSense provides a means for analyzing an applications privacy collecting functions without having to understand the entirety of the code.** The students found the majority of results easy to evaluate whether or not they were false positives. While not all of their determinations may be accurate, this does point to the fact that for the most part, not many files had to be reviewed. PrivSense at the very least provides a good starting point for reviewing the collection of privacy sensitive data within an app. Without using PrivSense or something similar, an analyst would have a much harder

time finding all of the instances of an application collecting privacy sensitive data. It would take an analyst much longer to review and understand all of the source code files and annotate potential areas of private data collection.

From **2**: **28% of the identifiers reviewed that were related to private data collection were not expected.** Some examples being; a fitness tracking app collecting user education information, and a beauty app collecting GPS location data. In the later example the student stated in her notes that she tried to find why location data was used and was unable to do so. While this statistic may differ after more identifiers are reviewed, it shows that applications collect data that might surprise users. Additional work could be done to compare identifiers reported with the privacy policy of an application to determine if the applications disclose all of the information they collect.

From **3**: **Most applications are able to be decompiled by free open source software to the point that an accurate analysis can be performed.** If the level of obfuscation was higher, an automated analysis by a tool such as PrivSense would require far more manual verification. Additionally, this shows that the majority of applications do not intentionally obfuscate their code so the presence of such obfuscation should raise a red flag about its intentions.

4.3.2 False Negative Analysis.

To check for identifiers that are indicative of personal information gathering that were not picked up by PrivSense could have required manually investigating the full source code. Instead of performing such an extensive search PrivSense was used

with reduced thresholds for Levenshtein and semantic similarity. By reducing the thresholds to .5 and .4 respectively, PrivSense produced a result set much larger but without including as many obviously irrelevant results. One app from each category was randomly selected and a result set was produced for each with the reduced thresholds. Each result set was manually compared against its respective result set for the same app produced by PrivSense with the normal thresholds.

Table 4.2 shows the difference between the two sets across all categories that had at least one relevant new result. The "Relevant New Results" column shows the amount of the total results that were considered actually relevant to private information collection. The "Unique Relevant New Results" column shows the relevant results when similar identifiers were combined into one result. For example, the identifiers `getCreditCardNumber` and `CreditCardNumber` can be considered as code that potentially collects the same private information of a credit card number so those two would be collapsed into one result.

Only five of 35 apps had greater than ten unique relevant false negatives, thirteen apps had five or less, and six apps had only one relevant unique result. The greatest number of false negatives was 46 for the 'Expedia' app which also had the highest number of total new results. The 'Expedia' app is in the 'Travel and local' category for which the results would generally be related to location information, contact information, and personal details about family in order to book trips. Since there are many privacy sensitive keywords that are relevant to a travel booking app it makes sense that there would be more results in general and therefore more false negatives. PrivSense could be tuned manually to lower thresholds to account for

categories expected to have a greater number of results such as 'Travel and Local' as well as for 'Dating' and 'Finance' which were the categories that had the two next highest unique relevant false negatives.

Table 4.2: False Negative Analysis.

App category	**App name**	**Relevant New Results**	**Unique Relevant New Results**
Shopping	ebates	31	12
Books and reference	google books	1	1
Art and design	mandala coloring book	2	1
Finance	credit-sesame	63	18
Game	roblox client	6	5
Android wear	google music	4	3
Productivity	microsoft office powerpoint	1	1
Video players	kinemaster free	1	1
Weather	weather radar widget	3	2
Food and drink	panera bread	31	12
Social	grindr	69	10
Tools	easyMover	10	9
Education	wordbit enes	1	1
Medical	szyk myheart	4	4
Music and audio	amazon mp3	11	8
Family	facebook talk	4	3
House and home	wifitv tvremote	5	4
Travel and local	expedia bookings	164	46
Maps and navigation	parkwhiz driverApp	20	9
Libraries and demo	hdmi app	2	1
Events	gaylord wayfinding	15	10
Health and fitness	dothesplits	19	8
Business	adobe scan	18	9
Personalization	panda emoji theme	5	3
Dating	jaumo casual	58	26
Communication	google messaging	14	5

Table 4.2: False Negative Analysis.

App category	**App name**	**Relevant New Results**	**Unique Relevant New Results**
Beauty	piupiuapps hairstyles	2	2
Parenting	amila parenting	11	7

CHAPTER 5

DISCUSSION AND FUTURE WORK

False positives and negatives. To reduce false positives, PrivSense pre-checks identifiers against a white list; a list of words known to have no privacy implications. In the future, as this list of non-private identifiers grows, the Levenshtein and semantic similarity thresholds could be increased to reduce false positives. Additional false positives are also caused by the fact that the Android package structure can be organized arbitrarily by a developer. It is difficult to discern if code belongs to the app or third parity libraries. We opted to only extract identifiers from the source contained under the main package sub-directory of a decompiled app. This avoids scanning commonly used third party libraries multiple times, focuses on the main activities of the app, and reduces false positives. The results produced by PrivSense therefore represent a *lower bound* of privacy identifiers found in the source.

False negatives can be caused by several factors: identifiers could be named arbitrarily, obfuscated, and also due to decompilation errors. The number of false negatives could be reduced by decreasing the thresholds for Levenshtein or semantic similarity detection although that would potentially be at the cost of increasing false positives. As discussed in Section 4 PrivSense could select different thresholds for the two similarity algorithms based on a categories high probability of collecting privacy sensitive data such as *Finance*, *Travel*, or *Dating*. Moreover, even when code is obfuscated, the program structure still remains. We plan to incorporate more sophisticated learning algorithms to address code obfuscation in the future.

Human perception about privacy. Even though the work of Markos et al. [12] and Schomaker et al. [13] is based on cross-sectional human subject studies in several countries, their scoring may be affected by several factors. First, the subject population was biased, e.g., the majority of the participants in [13] were women (59.1%). Second, participants may not have a good understanding about privacy, which is related to a person's level of Internet experience [30]. Therefore, the scores in Table 3.1 may not be a perfect measure for sensitivity level. But to the best of our knowledge, our work is the first to address consumers' perception in Android privacy research.

Future work. A limitation for PrivSense is its inability to analyze identifiers that are concatenated phrases. An example being *userCreditCardExpirationDate*, where it is clearly a concatenation of five words. This specific example would not have any semantic similarity since it is in itself not an English word and would likely not have a high enough Levenshtein similarity to be recorded as a finding by PrivSense since it is so long. An obvious next step for PrivSense would be to include a step between identifier extraction and similarity analysis that normalizes identifiers and tries to deconstruct identifiers such as the example given by separating words out by camel case, underscores or dashes. Once an identifier has been normalized in this way,F each individual word found within it could be analyzed separately for similarity to the privacy related keywords. Additionally, the semantic similarity analysis portion of PrivSense could be improved to detect meaning within phrases in addition to the meaning of single words before comparing to meaning of the privacy related keywords.

Currently, PrivSense weights detected identifiers without considering the syntactic elements (classes, methods, variables, etc.) that the identifier was extracted from. However, syntactic elements could provide insights into the scope of the privacy identifier. For example, if an identifier `religious_affiliation` was a class name, it could have more significant privacy implications than if it was a variable name. We plan to incorporate additional weighting based on the syntactic elements to improve the current scoring scheme.

Apps can collect private information through the permission system, or through the user interface (e.g., a text field that prompts a user for social security number). Most of previous research focuses on analyzing the permission system. However, the relationship of identifiers to their syntactic elements allows us to identify UI elements that trigger such data collection outside the scope of permissions. As a result, we could detect what UI actions taken by a user would cause private information leakage.

While PrivSense is well suited for large-scale analysis, it is not yet capable of analyzing code in other languages. It was demonstrated that 37% of Android apps contain at least one method or activity that is executed natively in C/C++ [31]. Therefore, our future work includes extending PrivSense to languages beyond Java.

Finally, extending PrivSense to compare the reported privacy related identifiers to an applications given privacy policy would allow for the presentation of sensitive data collected without the users awareness.

The data set and source code used for the research presented in this thesis can be found on Github at `https://github.com/whavey/android_research`.

CHAPTER 6

CONCLUSION

This work proposes PrivSense, a tool that extracts privacy identifiers from Android apps, and weighs them by sensitivity as perceived by consumers. With PrivSense, one can estimate the amount of private information an application may be processing. Through analyzing 925 apps across 35 categories, we found apps that perform superfluous data collection or belong to the wrong category.

BIBLIOGRAPHY

[1] "How big data can drive better customer service." Accessed July 27, 2020, https://freshdesk.com/customer-support/big-data-blog/.

[2] "Tracking phones, Google is a dragnet for the police." Accessed July 27, 2020, https://www.nytimes.com/interactive/2019/04/13/us/google-location-tracking-police.html.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Acm Sigplan Notices*, vol. 49, pp. 259–269, ACM, 2014.

[4] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pp. 93–104, ACM, 2012.

[5] P. P. Chan, L. C. Hui, and S.-M. Yiu, "Droidchecker: analyzing android applications for capability leak," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pp. 125–136, ACM, 2012.

[6] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs.," in *NDSS*, p. 234, 2013.

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pp. 239–252, ACM, 2011.

[8] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 627–638, ACM, 2011.

[9] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pp. 317–326, ACM, 2012.

[10] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie, "Uiref: analysis of sensitive user inputs in android applications," in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 23–34, ACM, 2017.

[11] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, "Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps.," in *NDSS*, 2018.

[12] E. Markos, G. R. Milne, and J. W. Peltier, "Information sensitivity and willingness to provide continua: a comparative privacy study of the united states and brazil," *Journal of Public Policy & Marketing*, vol. 36, no. 1, pp. 79–96, 2017.

[13] E.-M. Schomakers, C. Lidynia, D. Müllmann, and M. Ziefle, "Internet users' perceptions of information sensitivity–insights from germany," *International Journal of Information Management*, vol. 46, pp. 142–150, 2019.

[14] E. Okoyomon, N. Samarin, P. Wijesekera, A. Elazari Bar On, N. Vallina-Rodriguez, I. Reyes, Á. Feal, and S. Egelman, "On the ridiculousness of notice and consent: Contradictions in app privacy policies," 2019.

[15] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. 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, (New York, NY, USA), p. 3–14, Association for Computing Machinery, 2011.

[16] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, (New York, NY, USA), p. 93–104, Association for Computing Machinery, 2012.

[17] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pp. 539–552, 2012.

[18] K. Gudeth, M. Pirretti, K. Hoeper, and R. Buskey, "Delivering secure applications on commercial mobile devices: The case for bare metal hypervisors," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, (New York, NY, USA), p. 33–38, Association for Computing Machinery, 2011.

[19] X. Chen, H. Huang, S. Zhu, Q. Li, and Q. Guan, "Sweetdroid: Toward a context-sensitive privacy policy enforcement framework for android os," in *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*, WPES '17, (New York, NY, USA), p. 75–86, Association for Computing Machinery, 2017.

[20] Y. Chen, M. Zha, N. Zhang, D. Xu, Q. Zhao, X. Feng, K. Yuan, F. Suya, Y. Tian, K. Chen, *et al.*, "Demystifying hidden privacy settings in mobile apps," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 570–586, IEEE, 2019.

[21] Q. Wang, H. Xue, F. Li, D. Lee, and B. Luo, "# donttweetthis: Scoring private information in social networks," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 4, pp. 72–92, 2019.

[22] "Levenshtein distance." Accessed July 27, 2020, `https://en.wikipedia.org/wiki/Levenshtein_distance`.

[23] "Semantic similarity." Accessed July 27, 2020, `https://en.wikipedia.org/wiki/Semantic_similarity`.

[24] "Instant pot." Accessed July 27, 2020, `https://play.google.com/store/apps/details?id=com.instantbrands.app&hl=en_US`.

[25] "Taimi – lgbtqi+ dating, chat and social network." Accessed July 27, 2020, `https://play.google.com/store/apps/details?id=com.takimi.android&hl=en_US`.

[26] "Beijing's new national intelligence law: From defense to offense." Accessed July 27, 2020, `https://www.lawfareblog.com/beijings-new-national-intelligence-law-defense-offense`.

[27] "You may be asked to verify Wechat pay. here's how to do it." Accessed July 27, 2020, `https://www.thatsmags.com/china/post/27250/you-may-be-asked-to-verify-wechat-pay-here-s-how-to-do-it`.

[28] "Privacy policy." Accessed July 27, 2020, `https://rule.alibaba.com/rule/detail/2034.htm`.

[29] "Tiktok said to be under national security review." Accessed July 27, 2020, `https://www.nytimes.com/2019/11/01/technology/tiktok-national-security-review.html`.

[30] A. D. Miyazaki and A. Fernandez, "Consumer perceptions of privacy and security risks for online shopping," *Journal of Consumer affairs*, vol. 35, no. 1, pp. 27–44, 2001.

[31] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *The Network and Distributed System Security Symposium*, pp. 1–15, 2016.

APPENDICES

A Student Instructions

1. Review the results for apps generated from Privsense.

2. Find the identifiers specified in the results files for the respective app in its source code.

3. Annotate the identifier you are viewing as well as the privacy keyword it was similar to.

4. Review the code surrounding the identifier found and determine if it actually correlates to the collection of private data. Follow the code through multiple files/classes/methods if necessary.

5. Give a sensitivity rating (1-10) of the data being collected where 1 is least and 10 is most sensitive.

6. Annotate how difficult it was to verify that the identifier was or wasn't privacy sensitive with a rating of easy (immediately noticeable), medium (noticeable after reviewing some context in surrounding code), hard (had to review many other files/classes and make assumptions).

7. Annotate the source code files reviewed to verify privacy sensitivity (Give full paths).

8. Record if you see identifier names that are either unclear, or obfuscated i.e. have no clear structure or meaning (random characters). If obfuscation is present, record if you believe the obfuscation to be intentional by the developers in the Notes section. Please do this by best effort.

9. Do you expect the privacy sensitive data that you see to be used by this app and app category or do you find it unusual? Annotate with 'yes' for expected and 'no' for unusual.

10. Annotate any general notes you feel are relevant to share regarding how you determined the identifier was related to the collection of privacy sensitive data, or anything else you see fit.