UIPicker: User-Input Privacy Identification in Mobile Applications

Yuhong Nan [1], Min Yang [1], Zhemin Yang [1], Shunfan Zhou [1], Guofei Gu [2], and XiaoFeng Wang [3]

[1]School of Computer Science, Fudan University
[1]Shanghai Key Laboratory of Data Science, Fudan University
[2]SUCCESS Lab, Texas A&M University
[3]Indiana University at Bloomington
{*nanyuhong, m_yang, yangzhemin, 11300240020*}*@fudan.edu.cn*
guofei@cse.tamu.edu, xw7@indiana.edu

Abstract

Identifying sensitive user inputs is a prerequisite for privacy protection. When it comes to today's program analysis systems, however, only those data that go through well-defined system APIs can be automatically labelled. In our research, we show that this conventional approach is far from adequate, as most sensitive inputs are actually entered by the user at an app's runtime: in our research, we inspect 17, 425 top apps from Google Play, and find that 35.46% of them involve sensitive user inputs. Manually marking them involves a lot of effort, impeding a large-scale, automated analysis of apps for potential information leaks. To address this important issue, we present *UIPicker*, an adaptable framework for automatic identification of sensitive user inputs. UIPicker is designed to detect the semantic information within the application layout resources and program code, and further analyze it for the locations where security-critical information may show up. This approach can support a variety of existing security analysis on mobile apps. We further develop a runtime protection mechanism on top of the technique, which helps the user make informed decisions when her sensitive data is about to leave the device in an unexpected way. We evaluate our approach over 200 randomly selected popular apps on Google-Play. UIPicker is able to accurately label sensitive user inputs most of the time, with 93.6% precision and 90.1% recall.

1 Introduction

Protecting the privacy of user data within mobile applications (*apps* for short) has always been at the spotlight of mobile security research. Already a variety of program analysis techniques have been developed to evaluate apps for potential information leaks, either dynamically [19, 23, 41] or statically [15, 26]. Access control mechanisms [27, 22, 33, 17] have also been proposed to enforce fine-grained security policies on the way that private user data can be handled on a mobile system. These techniques are further employed by mobile app marketplaces like Google Play (e.g., Bouncer [7]) to detect the apps that conduct unauthorized collection of sensitive user data.

Identifying sensitive user inputs. Critical to those privacy protection mechanisms is the labeling of sensitive user data. Some of the data are provided by the operating system (OS), e.g., the GPS locations that can be acquired through system calls like `getLastKnownLocation()`. Protection of such information, which we call *System Centric Privacy* data, can leverage relevant data-access APIs to set the security tags for the data. More complicated here is the content the user enters to a mobile app through its user interface (UI), such as credit-card information, username, password, etc. Safeguarding this type of information, called *User-Input Privacy* (UIP) data in this paper, requires understanding its semantics within the app, before its locations can be determined, which cannot be done automatically using existing techniques.

Just like the system-controlled user data (e.g., GPS), the private content entered through the UI is equally vulnerable to a variety of information-leak threats. It has been reported [5, 10, 4, 6] that adversaries can steal sensitive user inputs through exploiting the weaknesses inside existing protection mechanisms. For example, fraud banking apps to steal user's financial credentials with very similarity UIs. Besides, less security-savvy developers often inadvertently disclose sensitive user data, for example, transmitting plaintext content across public networks, which subjects the apps to eavesdropping attacks. Recent work further shows that side channels [18] and content-pollution vulnerabilities [42] can be leveraged to steal sensitive user inputs as well. In our research, we found that among 17,425 top Google-Play apps, 35.46% require users to enter their confidential information.

Given its importance, UIP data urgently needs protec-

tion. However, its technical solution is by no means trivial. Unlike system-managed user data, which can be easily identified from a few API functions, sensitive user inputs cannot be found without interpreting the context and semantics of UIs. A straightforward approach is to mark all the inputs as sensitive [15], which is clearly an overkill and will cause a large number of false positives. Prior approaches [43, 15, 36, 38, 40] typically rely on users, developers or app analysts to manually specify the contents within apps that need to be protected. This requires intensive human intervention and does not work when it comes to a large-scale analysis of apps' privacy risks.

To protect sensitive user inputs against both deliberate and inadvertent exposures, it is important to automatically recognize the private content the user enters into mobile apps. This is challenging due to the lack of fixed structures for such content, which cannot be easily recovered without analyzing its semantics.

Our work. To address this issue, we propose our research *UIPicker*, a novel framework for automatic, large-scale User-Input Privacy identification within Android apps. Our approach leverages the observation that most privacy-related UI elements are well-described in layout resource files or annotated by relevant keywords on UI screens. These UI elements are automatically recovered in our research with a novel combination of several natural language processing, machine learning and program analysis techniques. More specifically, UIPicker first collects a training corpus of privacy-related contents, according to a set of keywords and auto-labelled data. Then, it utilizes the content to train a classifier that identifies sensitive user inputs from an app's layout resources. It also performs a static analysis on the app's code to locate the elements that indeed accept user inputs, thus filtering out those that actually do not contain private user data, even though apparently they are also associated with certain sensitive keywords, e.g., a dialog box explaining how a strong password should be constructed.

Based on UIPicker, we further develop a runtime privacy protection mechanism that warns users whenever sensitive data leave the device. Using the security labels set by UIPicker, our system can inform users of what kind of information is about to be sent out insecurely from the device. This enables the user to decide whether to stop the transmission. UIPicker can be used by the OS vendors or users to protect sensitive user data in the presence of untrusted or vulnerable apps. It can be easily deployed to support any existing static and dynamic taint analysis tools as well as access control frameworks for automatic labeling of private user information.

To the best of our knowledge, UIPicker is the first approach to help detect UIP data in a large scale. Although the prototype of UIPicker is implemented for Android, the idea can be applied to other platforms as well. We implemented UIPicker based on FlowDroid [15] and built our identification model using 17,425 popular Google Play apps. Our evaluation of UIPicker over 200 randomly selected popular apps shows that it achieves a high precision (93.6%) and recall (90.1%).

Contributions. In summary, this paper makes the following contributions.

- We measure the distribution of UIP data based on 17,425 classified top free applications from different categories. The results show that in some categories, more than half of applications contain UIP data. Further protection of these UIP data is in urgent need.
- We propose UIPicker, a series of techniques for automatically identifying UIP data in large scale. Lots of existing tools can benefit from UIPicker for better privacy recognition in mobile applications.
- Based on UIPicker, we propose a runtime security enhancement mechanism for UIP data protection, which helps user to make informed decisions when such data prepare to leave the device with insecure transmission.
- We conduct a series of evaluation to show the effectiveness and precision of UIPicker.

Roadmap. The rest of this paper is organized as follows. Section 2 gives the motivation, challenges and identification scope of UIP data, then introduces some background knowledge about Android layout resources. Section 3 gives an overview of UIPicker and illustrates the key techniques applied for identifying UIP data. Section 4 describes the identification approach step by step. Section 5 describes the runtime security enhancement framework based on UIPicker's identification results. Section 6 gives some implementation details about UIPicker. Section 7 gives evaluation and Section 8 discusses the limitation of UIPicker. Section 9 describes related work, and Section 10 concludes this work.

2 Problem Statement

In this section, we first provide a motivating example of users' sensitive input in two UI screens, then we investigate challenges in identifying such data and clarify our identification scope of UIP data. We also give some background knowledge about Android layout resources for further usage.

2.1 Motivating Example

Figure 1 shows two UI screens that contain some critical sensitive information in the Amazon Online Store [1]

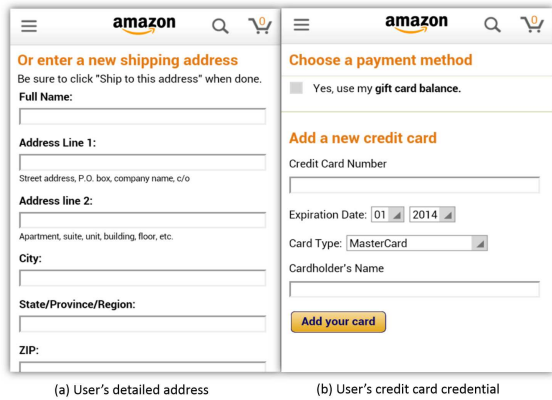(a) User's detailed address   (b) User's credit card credential

Figure 1: Examples of User-Input Privacy (UIP) Data

app. In Figure 1(a), the user is required to input his/her detailed address for delivering products. Figure 1(b) requires user to input the credit card credential to accomplish the payment process. Many apps in mobile platform would require such sensitive data for various functional purposes. Most of such data are personal information that users are unwilling to expose insecurely to the public.

Although UIP data can be highly security-sensitive and once improperly exposed, could have serious consequences, little has been done so far to identify them at a large scale. The key issue here is how to automatically differentiate sensitive user inputs from other inputs. In our research, we check the top 350 free apps on Google Play, and find that on average each of them contains 11 fields across 6 UI screens to accept user inputs; however many of these fields do not accommodate any sensitive data. Static analysis tools like FlowDroid [15] only provide options to taint all user inputs as sensitive sources (e.g. *Element.getText()*). Analyzing in this way would get fairly poor results because sensitive user inputs we focus are mixed in lots of other sources we do not care. Such problem also exists in runtime protection on users' sensitive inputs. For example, in order to prevent sensitive user inputs insecurely leaking out, an ideal solution would be warning users when such data leave the device. Alerting all user inputs in this way would greatly annoy the users and reduce the usability because many normal inputs do not need to be treated as sensitive data.

2.2 Challenges

UIP data can be easily recognized by human. However, it is quite challenging for the machine to automatically identify such data with existing approaches in large-scale.

First, UIP data can not be identified through runtime monitoring. As these sensitive data are highly unstructured, they can not be simply matched by regex expressions when users input them. Besides, like any normal inputs, privacy-related inputs are sparsely distributed in various layouts in a single app, and most UI screens contain such private data require login or complex trigger conditions, which makes it very difficult for automatic testing tools like [8, 34] to traverse such UI screens exhaustively without manual intervention.

Identifying UIP data by traditional static analysis approaches is also impractical. In program code's semantic, sensitive input does not have explicit difference compared to normal input. Specifically, all of such input data can be accepted by apps, then transmitted out or saved in local storage in the same way, which makes it difficult to distinguish them through static analysis approaches.

UIPicker identifies UIP data in apps from another perspective, it analyzes texts describing sensitive inputs other than data themselves. This is because texts in UI screens usually contain semantic information that describes the sensitive input. Besides, layout description texts in layout files also contain rich semantic information to reveal what the specific element is intended to be in the UI screen by developers. UIPicker is primarily designed to help identify UIP data in *benign* apps. The identification results can be further used for security analysis or protection of users' sensitive data. Note that in this work we do not deal with malicious apps that intentionally evade our analysis, e.g., malware that constructs its layout dynamically or uses pictures as labels to guide users to input their sensitive data.

2.3 Identification Scope

UIP data could be any piece of data that users consider to be sensitive from inputs. In the current version of UIPicker, we consider the following 3 categories as they cover most existing UIP data in current apps:

- **Account Credentials and User Profiles:** Information that reveals users' personal characters when they login or register, which includes but not limited to data such as username, user's true name, password, email address, phone number, birth date.
- **Location:** Plain texts that represent address information related to users. Different from system derived location (latitude and longitude), what we focus here is location data from users' input, e.g., the delivering address in shopping apps or the billing address for credit cards.
- **Financial:** Information related to users' financial activities, e.g., credit card number, expire date and security code.

The objective of UIPicker is to automatically identify such data from app resources in large-scale. Note

that UIP data might not be limited to items listed here. UIPicker is capable of expanding its identification scope easily, as further discussed in Section 4.2.

2.4 Android Layout Background

Here we give some background knowledge about Android layout resources which UIPicker will use in our identification approach.


```
PackageName: com.Amazon.android

res/values/public.xml

<resources>
 <public type="id"
  name="opl_credit_card_type"
  id="0x7f0e0352" />
 <public type="id"
  name="opl_credit_card_type_label"
  id="0x7f0e0353" />
</resources>

res/values/strings.xml

<resources>
 ......
 <string name=
  "opl_new_payment_credit_card_type_title">
     Card Type</string>
 <string name=
  "opl_new_payment_credit_card_name">
     "Cardholder's Name"</string>
</resources>

res/layout/add_credit_card.xml

<LinearLayout>
 <TextView android:id="@id/opl_credit_card_type_label"
       android:text="@string/opl_new_payment_credit_card_type_title" />
 <Spinner android:id="@id/opl_credit_card_type"
       android:prompt="@string/opl_new_payment_credit_card_select_type" />
 <TextView android:text="@string/opl_new_payment_credit_card_number" />
 <EditText android:id="@id/opl_credit_card_number" android:inputType="number" />
 <TextView android:text="@string/opl_new_payment_credit_card_expiration_date" />
 <LinearLayout>
     <Spinner android:id="@id/opl_credit_card_expiration_month"
           android:prompt="@string/
opl_new_payment_credit_card_expiration_date_month" />
     <Spinner android:id="@id/opl_credit_card_expiration_year"
           android:prompt="@string/
opl_new_payment_credit_card_expiration_date_year" />
 </LinearLayout>
 <TextView android:text="@string/opl_new_payment_credit_card_name" />
 <EditText android:id="@id/opl_credit_card_name" />
 <Button android:id="@id/opl_credit_card_save"
       android:text="@string/opl_new_payment_credit_card_save_button" />
</LinearLayout>
```
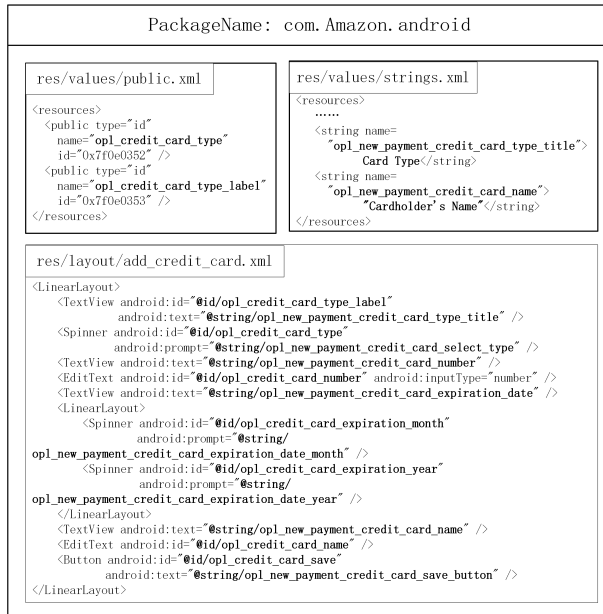
Figure 2: Android Layout Description Resources

Layout resources define what will be drawn in the UI screen(s) of the app. In Android, a User Interface is made up of some basic elements (e.g., *TextView*, *EditText*, *Button*) to display information or receive input. Android mainly uses XML to construct app layouts, thus developers can quickly design UI layouts and screen elements they wish to contain, with a series of elements such as buttons, labels, or input fields. Each element has various attributes or parameters which are made up of name-value pairs to provide additional information about the element.

Android layout resources are distributed in different folders in the app package. Layout files for describing UI screens are located in folder *res/layout*. The unique hex digit IDs for identifying each element in layout files are in *res/value/public.xml* and texts showed in UI screens to users are in *res/values/strings.xml*. Resources in *res/values/* are referenced by texts with specific syntax (e.g. @*String,* @*id*) in the layout files in *res/layout* for ease of development and resource management.

Figure 2 shows some layout resources used for constructing the UI in Figure 1(b). The entry is a layout file named *add_credit_card.xml*. It contains two *EditText* elements to accept the credit card number and the card holder's name, three *Dropdown list* elements (named as *spinner* in Android) to let user select card type and expiration date. In the *EditText* for requesting the card number, it uses @*id/opl_credit_card_number* to uniquely identify this element for the app. Syntax like *android:inputType=number* suggests that this *EditText* only accepts digital input. There is also a *TextView* before *EditText* with attribute *android:text=@string/opl_new_payment_credit_card_number*, which means the content showed in this label will be string referenced to *opl_new_payment_credit_card_number* in */res/values/stings.xml*.

3 System Overview

In this section, we give an overview of UIPicker and describe the key techniques applied in our identification framework.

Overall Architecture. Figure 3 shows the overall architecture of UIPicker. UIPicker is made up of four components to identify layout elements which contain UIP data step by step. The major components can be divided into two phases: model-training and identification. In the model-training phase (Stage 1,2,3), UIPicker takes a set of apps to train a classifier for identifying elements contain UIP data from their textual semantics. In the identification Phase (Stage 1,3,4), UIPicker uses both the trained classifier (Stage 3) and program behavior (Stage 4) to identify UIP data elements.

Pre-Processing. In the Pre-Processing module, UIPicker extracts the selected layout resource texts and reorganizes them through natural language processing (NLP) for further usage. This step includes word splitting, redundant content removal and stemming for texts. Pre-Process can greatly reduce the format variations of texts in layout resources caused by developers' different coding practice.

Privacy-related Texts Analysis. For identifying UIP data from layout resources, the first challenge is how to get privacy-related texts. One can easily come up with a small set of words about UIP data, but it is very difficult to get a complete dictionary to cover all such semantics. In our case, leveraging an English dictionary like WordNet [14] for obtaining semantically related words is limited in the domain of our goals. Many words that are semantically related in privacy may not be semantically related in English, and many words that are semantically related in English may not appear in layout resource texts as well. For example, both "signup" and "register" represent to create a new account in an app's login screen, but
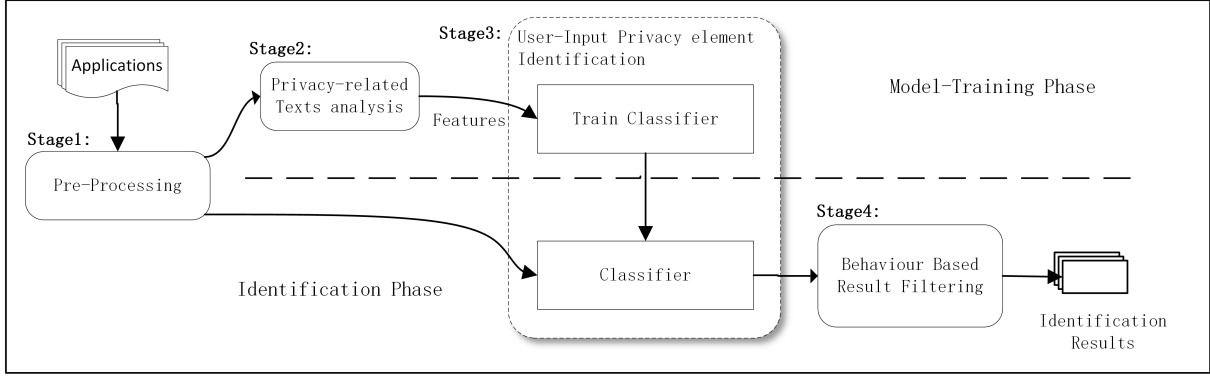
Figure 3: System Overview of UIPicker

they can not be correlated from a dictionary like Word-Net.

Besides, UIP data in apps are often described with a single word or very short phrases (e.g. "password" or "input password") in layout resources. Due to the lack of complete sentences describing UIP data, natural language processing techniques [35] like dependency relation pattern extraction is not suitable in our scenario.

UIPicker expands UIP semantic texts with a few privacy-related seeds based on a specific feature extraction approach. It first automatically labels a subset of layouts which could contain UIP data by heuristic rules, then extracts privacy-related semantics from such layouts by applying clustering algorithms. It helps us to automatically extract privacy-related texts with little manual effort. As a result, these inferred texts can be used as features for identifying whether an element is privacy-related or not in the next step.

UIP Data Element Identification. Based on the given set of privacy-related textual semantics from the previous step, to what extent an element contains privacy-related texts can be identified as sensitive? As previous work [37] showed, purely relying on keyword-based search would result in a large number of false positives. For example, sensitive item "username" could always be split into "user" and "name" as two words in apps, and none of the single word can represent "username". Besides, certain words like "address" have a confounding meaning. For instance, such phrase showed in a layout screen "address such problem" does not refer to location information.

In this step, UIPicker uses a supervised machine learning approach to train a classifier based on a set of semantic features generated in the previous stage. Besides, it fully takes the element's context in the whole layout into consideration for deciding whether the element is privacy-related or not. With this trained model, for any given layout element with description texts, UIPicker can

tell whether it is related to UIP from its textual semantics.

Behavior Based Result Filtering. Besides identifying elements that contain UIP data from their textual semantics, we also need to check whether a privacy-related element is actually accepting user input. In other words, we need to distinguish user inputs from other static elements such as buttons or labels for information illustration in layout screens. Although Android defines *Edit-Text* for accepting user input, developers can design any type of element by themselves (e.g. customized input field named as *com.abc.InputBox*). Besides, apps also receive user inputs in an implicit way through other system defined elements without typing the keyboard by users. For example, in Figure 1(b), the expire date of credit card is acquired by selecting digits from the *Spinner* element.

We observe that for each privacy-related element identified by UIPicker in the previous stage, the data should be acquired by the app with user's consent if it is actually accepting user input. For example, the user clicks a button "OK" to submit data he/she inputs. When reflected in the program code, the user input data should be acquired by the system under certain event trigger functions. We use static code analysis to check whether an arbitrary element can be matched with such behavior, thus filter out irrelevant elements we do not expect.

4 IDENTIFICATION APPROACH

In this section, we explain the details of four stages in UIPicker's identification approach.

4.1 Stage 1: Pre-Processing

Resource Extraction. We first decode the Android APK package with apktool [2] for extracting related resource files we need. Our main interest is in UI-related content, thus for each app, we extract **UI Texts** and **Layout Descriptions** from its decompiled layout files.

5

Layout Resource	Sample
UI Texts	Add a new credit card, Credit Card Number Expiration Date, Card Type, Cardholder's name
Layout Descrioptions	@id/opl_credit_card_number @string/opl_new_credit_card_expiration_date_month @string/opl_new_credit_card_save_buton

Table 1: Selected resources of Amazon Online's "Add Credit Card" screen

- **UI Texts**. UI texts are texts showed to users in the layout screen. In Android, most of such texts are located in */res/values/strings.xml* and referenced by syntax *@String/[UI text identifier]* in element's attribute. Some *UI texts* are directly written in layout files as attribute values of UI elements, e.g., *android:hint='Please input your home address here'*.
- **Layout Descriptions**. Layout Descriptions are texts only showed in layout files located in */res/layout/*. For these texts, we consider all strings starting with syntax *@id* and *@String* to reflect what the element is intended to be from their textual semantics.

The main difference between *UI texts* and *layout descriptions* is that *UI texts* are purely made up of natural language while *layout descriptions* are mainly name identifiers (both formatted and unformatted) with semantic information. As developers have different naming behaviors when constructing UIs, in most cases, semantic information in *layout descriptions* is more ambiguous than that in *UI texts*.

We extract these groups of resources for further analysis because these selected targets can mostly reflect the actual content of the app's layout. For example, the selected resources about Amazon's "Add Credit Card" screen in Figure 1(b) are showed in Table 1.

Word Splitting. Although most of *layout descriptions* are meaningful identifiers for ease of reading and program development, normally they are delimiter-separated words or letter-case separated words. For example, "phone number" can be described as "phone_number" or "PhoneNumber". Thus we split such strings into separated word sets. Besides, some of *layout descriptions* are concatenated by multiple words without any separated characters. For these data, we split them out by iteratively matching the maximum length word in WordNet [14] until the string cannot be split any more. For example, string "confirmpasswordfield" will be split into "confirm", "password", and "field".

Redundant Content Removal. For all *UI texts* we extracted, we remove non-English strings through encoding analysis. For each word, we also remove non-text characters from all extracted resources such as digits, punctuation. After this, we remove stop words. Stop

Figure 4: After Pre-Processing, texts in left are transformed into formats in right

words are some of the most common words like "the", "is", "have". We remove such contents because they can not provide meaningful help in our analysis process.

Stemming. Stemming is the process for reducing inflected (or sometimes derived) words to their stem, base or root form. Stemming is essential to make words such as "changed", "changing" all match to the single common root "change". Stemming can greatly improve the results of later identification processes since they reduce the number of words in our resources. We implement *Porter Stemmer* [11] with python NLTK module [9].

Figure 4 shows part of texts before and after preprocessing for Amazon's "Add credit card" layout file. As we can see, all texts concatenated by '_' are split into separated words, "edthomephonecontact" is split into "edt", "home", "phone" and "contact" instead. We also transform words like "forgot", "forget" into a single uniformed format as "forget".

4.2 Stage 2: Privacy-related Texts Analysis

In this stage, we use Chi-Square test [39] to extract privacy-related texts from a subset of specific layouts. The intuition here is that privacy-related words prefer to be correlated in specific UIs such as the login, registration or settings page of the app. If some words appear together in these UI, they are likely to have semantic relevance to users' sensitive information. Thus, we use such layouts to extract privacy-related texts in contrast to other normal layouts.

Chi-Square Based Clustering. Chi-Square (*Chi^2*) test is a statistical test that is widely used to determine whether the expected distributions of categorical variables significantly differ from those observed. Specifically in our case, it is leveraged to test whether a specific term on UI screens is privacy-related or not according to its occurrences in two opposite datasets (privacy-related or non privacy-related).

Here we choose *UI texts* rather than *layout descriptions* to generate privacy-related texts due to the follow-
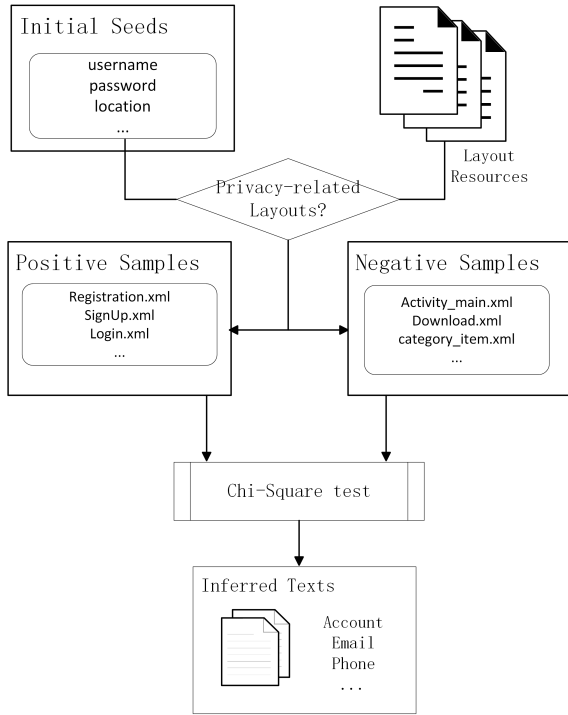
Figure 5: For each word assigned as the initial seed, UIPicker calculates its Chi-Square test for each word in positive samples and appends part of its top results into the privacy-related texts feature set.

ing two reasons: First, *layout descriptions* are not well structured as the naming behaviors vary very differently between apps (or developers), while *UI texts* are in a relatively uniformed format, thus making it easy to extract privacy-related texts from them. For example, a layout requesting a user's password must contain term "password" in the UI screen, while in layout descriptions it could be text like "pwd", "passwd", "pass". Second, as layout descriptions aim for describing layout elements, it may contain too much noisy texts like "button", "text" which would bring negative impact to the privacy-related text extraction.

Figure 5 shows how UIPicker generates privacy-related texts. First, we give a few words that can explicitly represent users' sensitive input we focus (e.g., email, location, credit card), and we call them *initial seeds*. Each layout sample is made up of a set of *UI texts* in its layout screen. Then, the initial seeds will be used to identify whether a specific layout sample is privacy-related or not based on the following two patterns:

- Logical relation between sensitive noun phrase (initial seed) and verb phrase, e.g., the pair (save, password).
- Logical relation between possessive (mainly word "your") and sensitive noun phrase (initial seed),

e.g., the pair (your, username).

As such patterns strongly imply actions that the app is requesting the user's sensitive input, for those layout samples satisfying one of these two patterns, we label them as privacy-related (positive samples). On the other hand, for layout samples that do not contain any of texts in the pattern (both noun phrase and verb, possessive phase), we label them as negative samples. Note that we do not label those layouts only containing initial seeds as positive or negative because a single word is insufficient for us to identify whether the layout is privacy-related or not.

Based on the two classified sample sets, for all distinct words appearing in positive samples, we use Chi-Square test and rank their results in a descending order. As a result, texts with higher Chi-Square scores mean they are more representative as privacy-related, which can easily be picked up from the top-ranked words in the test results.

The following example explains our analysis approach for finding financial-related textual semantics. We set "credit card" as an initial seed, then the layout shown in Figure 1(b) will be identified as a positive sample because both "credit card" and verb phrase "add" are included in this layout. Thus in our dataset, other similar layouts will be labeled as positive if it requires users to input credit card information as well. As a result, the positive sample will include more texts such as "expire", "date", "year", "month", which are also related to financial credentials and ranked in top of the Chi-Square test results.

Noisy Text Removal. Although Chi-Square test aims to cluster privacy-related texts, it still unavoidably introduces some irrelevant texts into its clustering results. This is mainly because not all texts in privacy-related layout are necessarily related to privacy. In order to generate a highly precise cluster of privacy-related texts to eliminate false positives in the UIP data element identification process, we introduce a little manual effort here for filtering out such irrelevant texts from the clustering result. Since Chi-Square test already helps us extract texts that are most probably related to privacy, looking through such a list is quite simple and effortless.

Alternative Approaches. In our research, we compared the Chi-Square test with two popular alternatives, frequency based text extraction and TF-IDF [31], both of which are found to be less effective. They all bring in more irrelevant contents than the Chi-Square test, more susceptible to the limitation of the layout level samples, that is, privacy related UI screens often contain a lot of normal texts, which become noises in our sensitive term identification. Also, using these two approaches, we need to continuously adjust their thresholds for select-

ing privacy-related text when their sample sizes change. This can be avoided when using the Chi-Square test. Nevertheless, we acknowledge that there may exist other feature extraction mechanisms that could perform better, which is one of our future work.

4.3 Stage 3: UIP Data Element Identification

In this stage, we explain the details of our machine-learning approach, which automatically identifies UIP data elements based on their textual semantics. UIPicker uses supervised learning to train a classifier based on a subset of element samples with privacy-related semantic features. As a result, for a given unclassified UI element, this step could identify whether it is semantically privacy-related from its description texts.

Feature Selection. We use privacy-related texts inferred from the previous stage as features for the identification module. A single word alone usually does not provide enough information to decide whether a given element is privacy-related. However, all such features in combination can be used to train a precise classifier. The main reason why these features work is that both *UI texts* and *layout descriptions* do in fact reveal textual semantic information, which a machine learning approach such as ours can discover and utilize. Note that in *layout descriptions*, it is often the case that developers use text abbreviations for simplicity when naming identifiers. For example, "address" in layout descriptions could be "addr". For this, we construct a mapping list of such texts we visited during the manual analysis. Thus, for each word in layout descriptions, we transform the abbreviation into complete one if it is contained by any privacy-related texts.

Besides, we also take semantic features of layout structure into consideration: the texts of this element's siblings. We observe that many elements are described by texts in its siblings. For example, In Figure 1(b), most of input fields are described by static labels which contain privacy-related text as instructions for requesting user inputs. As a result, texts from sibling elements can bring more semantic information for better identification results.

The classifier works on a matrix organized by one column per feature (one word) and one row per instance. The dimension for each instance is the size of our feature set (the number of texts from the previous step). The additional column indicates whether or not this instance is a privacy-related element.

Training Data. Since text fields can have different input types for determining what kind of characters are allowed inside the field, Android provides the *android:inputType* attribute to specify what kind of char-

acters are allowed for *EditText*. For example, an element with *inputType* valued *textEmailAddress* means only email address is accepted in this input field. There are several input types explicitly reflect the element containing UIP data we focus on, which can be used as the training data of the identification module. We list such sensitive attribute values[1] in the first column of Table 2.

Privacy Category	Attribute Value
Account Credentials & User Profile	textEmailAddress textPersonName textPassword textVisiblePassword password/email/phoneNumber
Location	textPostalAddress

Table 2: Sensitive attribute values in layout descriptions

The training data is constructed as follows: First, we automatically label all elements with sensitive attributes as positive samples since they are a subset of UIP data elements. We further manually label a set of elements involving financial information from the category "Financial" because such elements are covered by sensitive attributes Android provides. Besides, a set of negative samples are picked out through human labeling after filtering out the elements that contain any of the privacy-related texts we generated in Stage 2.

Classifier Selection. We utilize the standard support vector machine (SVM) as our classifier. SVM is widely used for classification and regression analysis. Given a set of training examples with two different categories, the algorithm tries to find a hyper-plane separating the examples. As a result, it determines which side of hyper-plane the new test examples belong to. In our case, for an unclassified unknown layout element with corresponding features (whether or not containing privacy-related texts extracted in the previous step) the classifier can decide whether it contains UIP data or not from its textual semantics.

4.4 Stage 4: Behavior Based Result Filtering

As a non-trivial approach for identifying UIP data, for each element identified as privacy-related from its layout descriptions, UIPicker inspects the behaviors reflected in its program code to check whether it is accepting user inputs, thus filtering out irrelevant elements from the identification results in the previous step.

---

[1]In some older apps, developers also use specific attribute like "android:password=True" to achieve the same goal as *inputType*. We list them in Table 2 and call them sensitive attribute values as well for simplicity.
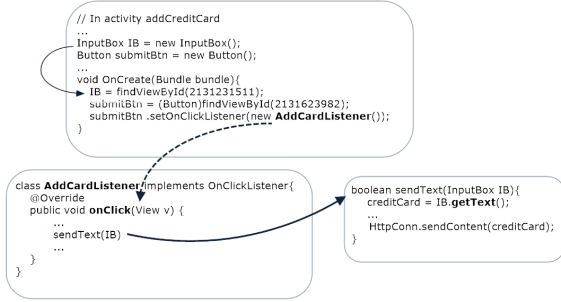
Figure 6: Sample codes for requesting a user's credit card number

User input data is generated based on a user's interactions with the app during runtime. In other words, the data will be acquired by the app under the user's consent. In Android, to get any data from a UI screen is achieved by calling specific APIs. Getting such data under user consent means these APIs are called under user-triggered system callbacks. For example, code fragments in Figure 6 shows the behavior reflected in the program when the app gets the user's credit card number in Figure 1(b). Here, the input field *IB* is defined by *IB=findViewById(21...1)* in activity *addCreditCard*. When the user clicks the "Add your card" button, in the program code, the *OnClick()* function in class *AddCardListener()* will be triggered by pre-registered system callback *submitBtn.setOnClickListener()*. Then, it invokes *sendText(IB)*, which sends the inputBox's object by parameter, and finally gets the user's card number by *IB.getText()*. One might consider why don't catch UIP data simply by checking whether the element is invoked by *getText()* API. The reason is that sometimes developers may also get values from UI screens like static text labels as well as user inputs, resulting in false negatives for our identification approach.

5 Runtime Security Enhancement with UIPicker

The security implications about UIP data are rooted from the fact that users have to blindly trust apps when they input sensitive data. With the help of UIPicker differentiating UIP data from other normal inputs, we can use taint tracking techniques to trace users' sensitive inputs and enable users to make informed decisions with a pop-up window when such data insecurely leave the device, thus effectively mitigating the potential threats posed by apps.

For UIP data, we consider the following two situations as insecure and should inform users to let them decide whether to proceed or not.

Plain Text Transmission. We consider any piece of UIP data should not be transmitted in plain text. Such situation can be easily identified by checking if the tainted sink is HTTP connection in runtime.

Insecure SSL Transmission. Previous works [34] showed that a large number of apps implement SSL with inadequate validations (e.g., app contains code that allows all hostnames or accepts all certificates). Insecure SSL transmission could be more dangerous because they may carry over critical sensitive data in most cases. UIP data should not be transmitted in this way as well.

Since UIPicker is deployed in off-line analysis by customized system vendors, one can also check whether the apps have securely implemented SSL off-line at the same time. We integrate a static analysis framework named MalloDroid [20] with UIPicker to automatically check SSL security risks by evaluating the SSL usage in apps. As MalloDroid can only find broken SSL usage regardless what data is transmitted via this channel, we also use FlowDroid to check if there exists data/control flow intersections between UIP data sources and SSL library invocation sinks in the app, thus confirming whether the UIP data in the app will be transmitted with security risks.

6 IMPLEMENTATION

Dataset. We crawled apps from Google Play Store based on its pre-classified 35 categories in Oct. 2014. For each category, we downloaded the top 500 apps. Excepting some connection errors occurred in the crawling process, totally we collected 17,425 apps as our dataset. This dataset will be used in both model training and evaluation of UIPicker.

Identification Approach. We implement the prototype of UIPicker as a mix of Python Scripts and Java code. The first three steps of UIPicker are developed using Python with 3,624 lines of code (LOC). The last step, static analysis for result filtering, is implemented in Java, which extends FlowDroid[15] and introduces additional 985 LOCs. All experiments are performed on a 32 core Debian server with Linux 2.6.32 kernel and 64GB memory.

For privacy-related text analysis, the initial seeds are assigned as texts in the second column of Table 3 for each privacy category. For each initial seed, we run the Chi-Square test using apps in our dataset. Since Android allows developers to use nested layout structures for flexibility, we also group sub-layout *UI texts* into their root layouts. For each round, we collect the top 80 words from the test results, this threshold is determined by balancing between the number of privacy-related terms that can be detected and the amount of noisy text introduced. After 7 (7 initial seeds) rounds of the Chi-Square test,

we collect 273 words from layout samples (some texts are overlapped in different round of Chi-Square test). We then remove 45 words as the noisy text by manual analysis within less than 3 minutes. As a result, UIPicker extracts 228 privacy-related terms from 13,392 distinct words. We list part of them in the third column of Table 3 corresponding to the privacy category they belong to. Such data are used as features for privacy-related element identification in the follow-up step.

Privacy Category	Initial Seeds	Representative Inferred Texts(Stemmed)
Login Credenticals &User Profile	username, password, email	mobil phone middl profile cellphon account nicknam firstnam lastnam person birth login confirm detail regist
Location	address, location	zip citi street postal locat countri
Financial	credit card, bank	secur month date pay year bill expir debit transact mm yy pin code

Table 3: Initial seeds and part of inferred privacy-related texts from Chi-Square test

The SVM classifier is implemented with scikit-learn[13] in poly kernel. We optimize the classifier parameters (gamma=50 and degree=2) for performing the best results.

For each element identified as privacy-related by the machine learning classifier, UIPicker conducts static taint analysis using FlowDroid[15] to check whether it satisfies specific behavior described in Section 4.4. Since FlowDroid successfully handles android life cycle (system event based callbacks) and UI widgets, the dataflow results should be both precise and complete. We set FlowDroid's layout mode as "ALL" to get each element's propagation chain that starts with function *findViewById([elementId])* and ends in *getText()*. As a result, for any element's info-flow path which contains system event function like *OnClick()*, the element can be identified as accepting user input.

Runtime Enhancement For each app, we use a list of elements containing UIP data identified from UIPicker with their unique IDs as the taint sources of TaintDroid[19] build in Android 4.1. Since Taint-Droid allows 32 different taint markings through a 32-bit bitvector to encode the taint tag, for those UIP data elements involved in insecure SSL usage, we label them as "SSL Insecure" in the taint source list, thus provide warnings to users when such data leave the device as well. We add a pop-up window for showing the leaked information to users when sensitive data leave the device. Our modification to TaintDroid is implemented with 730 LOCs in total.

7 Evaluation

In this section, we present our evaluation results. We first show the performance of UIPicker in Section 7.1, then we discuss its effectiveness and precision in Section 7.2 and Section 7.3. Then we evaluate our runtime security enhancement mechanism in Section 7.4.

7.1 Performance

During our experiment, the training phase of the classifier takes about 2.5 hours on average, the identification phase for the whole dataset takes 30.5 hours (6.27 seconds per app). Pre-Processing time for apps is included in both of these two phases. The static analysis for behaviour based result filtering is proceeded in 32 threads concurrently. Since UIPicker mainly targets for customized system vendors or security analysts, we consider such overhead quite acceptable.

7.2 Effectiveness

UIP Data Distribution. We show the general identification results of UIPicker in Table 4. In 17,425 apps, UIPicker finds that 6,179 (35.46%) contain UIP data. We list our results in a descending order of the identified total app amounts. As we can see, in 9 out of 35 categories, more than half of apps contain UIP data.

We make the following observations from this table. First, application categories such as BUSINESS, FINANCE, SHOPPING, COMMUNICATION and SOCIAL are more likely to request Account Credentials and User Profile information, which showed that these apps are closely related to users' personal activities. APP_WIDGETS (54.08%) is also ranked among top of the table. It is a set of apps which have small UIs embedded in the home screen of the device, e.g., Facebook, Youtube, Twitter. Since most of such apps provide login and account-specific functions, they prefer to request more UIP data as well. The SHOPPING category contains many location-related elements (1,605, 37%) because the delivering address are always generated from user inputs. It is also reasonable that both FINANCE and SHOPPING apps require many financial-related sensitive inputs. We believe such apps containing rich UIP data should be treated more carefully in both developing and security vetting process in order to make sure that sensitive data are well protected in both transmission and storage.

Comparative Results. We illustrate the effectiveness of UIPicker from two aspects. First, UIPicker identifies privacy data that system defined APIs do not touch but still be sensitive to users. Second, UIPicker achieves far better coverage than simply identifying UIP data by specific sensitive attribute values from the Android design specification.

Comparison with System Defined Sensitive APIs. As previously mentioned, specific sensitive resources

Application Category	Account Credentials & User Profile		Location		Financial		Total	
	#element	%app	#element	%app	#element	%app	#element	%app
BUSINESS	4,314	61.52%	1,112	38.28%	399	18.04%	5,825	62.73%
WEATHER	1,102	46.18%	1,086	59.24%	32	3.01%	2,220	62.45%
FINANCE	4,821	50.90%	1,106	33.47%	1,815	30.46%	7,742	55.31%
COMMUNICATION	2,756	53.83%	439	21.77%	213	14.31%	3,408	55.24%
SHOPPING	3,380	51.80%	1,605	37.00%	609	24.80%	5,594	54.60%
APP_WIDGETS	3,161	51.22%	816	31.43%	352	15.71%	4,329	54.08%
NEWS_AND_MAGAZINES	1,994	47.38%	529	34.68%	133	12.50%	2,656	54.03%
SOCIAL	2,889	52.62%	555	27.42%	146	8.27%	3,590	54.03%
TRAVEL_AND_LOCAL	2,826	49.00%	1,494	41.16%	452	16.87%	4,772	52.21%
PRODUCTIVITY	1,923	45.45%	394	18.59%	113	9.29%	2,430	48.69%
LIFESTYLE	2,243	43.29%	853	28.66%	341	14.03%	3,437	45.29%
TRANSPORTATION	1,634	39.00%	750	28.60%	273	11.00%	2,657	44.60%
SPORTS_GAMES	2,023	41.70%	509	22.67%	151	6.68%	2,683	43.32%
MEDICAL	1,478	40.04%	302	15.49%	169	7.04%	1,949	40.24%
HEALTH_AND_FITNESS	1,795	39.56%	344	15.06%	165	8.43%	2,304	39.96%
MEDIA_AND_VIDEO	1,079	37.15%	170	13.05%	72	3.61%	1,321	38.55%
TOOLS	1,110	36.36%	252	16.16%	121	8.08%	1,483	38.38%
MUSIC_AND_AUDIO	1,053	37.20%	219	11.40%	91	3.20%	1,363	38.00%
PHOTOGRAPHY	1,008	26.65%	205	9.82%	122	5.21%	1,335	28.46%
ENTERTAINMENT	973	27.71%	249	9.24%	215	5.62%	1,437	28.31%
BOOKS_AND_REFERENCE	924	26.80%	213	9.80%	156	5.60%	1,293	27.40%
EDUCATION	1,753	20.68%	461	9.84%	83	5.02%	2,297	21.69%
COMICS	390	16.60%	84	4.00%	69	3.00%	543	17.20%
PERSONALIZATION	440	16.23%	77	3.85%	32	1.83%	549	16.43%
CARDS	360	14.20%	40	3.20%	58	4.60%	458	15.80%
GAME_WIDGETS	302	13.25%	17	2.01%	56	4.42%	375	13.45%
ARCADE	390	12.22%	66	3.61%	24	0.80%	480	12.42%
LIBRARIES_AND_DEMO	302	10.84%	89	3.61%	136	3.01%	527	11.24%
GAME_WALLPAPER	242	11.00%	21	2.00%	55	4.20%	318	11.00%
BRAIN	396	10.60%	102	4.00%	71	2.20%	569	10.80%
GAME	302	9.82%	53	3.81%	16	0.80%	371	10.22%
SPORTS	209	10.22%	26	1.40%	15	0.80%	250	10.22%
CASUAL	267	9.60%	23	2.60%	10	0.40%	300	9.60%
APP_WALLPAPER	187	6.25%	34	2.42%	20	1.61%	241	6.65%
RACING	82	4.60%	16	0.40%	20	0.60%	118	4.60%
TOTAL	50,108	30.59%	14,311	16.26%	6,805	7.57%	71,224	35.46%

Table 4: UIP data distribution. #element denotes the number of UIP data elements in each category by different privacy type. %app denotes the percentage of apps in which these elements appear (500 per category). The last column shows the total number of UIP data elements and apps that contain UIP data.

Privacy Category	Android System Defined APIs
Account Credentials & User Profile	android.tel...TelephonyManager getLine1Number() android.accounts.AccountManager getAccounts()
Location	and...LocationManager getLastKnownLocation() android.location.Location: getLongitude() android.location.Location: getLatitude()

Table 6: System defined sensitive APIs related to UIPicker's identification scope

such as phonenumber, account and location can be regulated by fixed system APIs which we list in Table 6. We compare the amount of UIPicker's identification results with Android system derived sensitive data, which can help us understand to what extent, system defined sensitive APIs are insufficient to cover users' privacy.

As Table 5 shows, in our dataset, 4,900 apps use system defined APIs for requesting Account Credentials and Profile Information while UIPicker identifies 5,330 (30.59%) apps containing UIP. UIPicker identifies 2,883 (16.26%) apps in the whole dataset that request location privacy data from user inputs. Besides, 1,318 (7.57%) apps request financial privacy data from users, and none of system defined APIs can regulate such data. In general, UIPicker identifies 6,179 (35.46%) apps containing at least one category of UIP data, which have been largely neglected by previous work in privacy security analysis and protection.

As Column 4 in Table 5 shows, there is some overlap between system defined APIs and UIP data (1,340 for Account Credentials & User Profile, 2,282 for Location respectively). For each app, we check whether it contains both the system defined APIs and the UIP data in the same privacy category, e.g., invoking the getLastKnown-Location() API and requesting address information from the user input of the same app. In some cases, the same piece data may come from either UI input or API call. For example, using a phone number as the login account of the app. However in most cases, the overlapped data in the same privacy category may come from different sources without overlapping in code paths. For example, the invocation of get-location APIs is used for realtime geographic locating, while some location input could be

Privacy Category	System Defined APIs (#Apps)			Elements with Sensitive Attribute Values (#Elements)		
	API	UIPicker	Overlap	InputType	UIPicker	Incremental
Account Credentials & User Profile	4,900	5,330	1,340	24,021	46,227	26,087
Location	15,221	2,883	2,282	941	14,311	13,370
Financial	-	1,318	-	-	6,353	-
Total	15,632	6,179	-	24,962	71,224	46,262

Table 5: We compare UIPicker's identification results with apps containing system defined sensitive APIs (column 2-4) and elements containing sensitive attribute values (column 5-7).

a shopping address for delivering goods. Since precisely analyzing which input element may overlap with system defined APIs requires additional information-flow analysis, which is beyond this paper's scope, we leave it as future work for measuring the relationship between these two types of sensitive data.

Comparing to sensitive attribute values. In Section 4.3, we use elements containing sensitive attribute values as part of training data for our identification module. However, they can only cover a portion of UIP data because they are not intended for this purpose. Here we compare the amount of UIPicker's identification results with elements containing sensitive attribute values to show the effectiveness of UIPicker.

As Table 5 shows, in general, UIPicker identifies 46,262 more UIP data elements than simply identifying them by sensitive attribute values (e.g. textPassword). Especially for the Location category, UIPicker identifies 14,311 elements, which is nearly 15 times more than simply identifying them based on attribute "textPostalAddress".

Types of UIP Elements. We list the identification results of UIP data elements other than *EditText* in Table 7. In general, UIPicker finds 18,403 (25.84%) elements other than *EditText* to accept users' sensitive inputs. It is interesting to note that UIPicker also finds a large portion of *TextView* as UIP data elements. In most cases, although data in *TextViews* are not editable, they could be generated by users from other layouts and dynamically filled in *TextView* later. For example, the data from previous steps of a registration form, or fetched from the server after users' login. There are 5,075 (7.13%) customized input elements and 1,962 (2.75%) dropdown lists (*Spinners*) containing UIP data. Type "Others" in table contains elements such as *RadioButton*, *CheckBox*.

7.3 Precision

For evaluating the precision of UIPicker, we perform the evaluation of classifier based on the machine-leaning dataset mentioned in Section 4. We also conduct a manual validation for two reasons. First, since the training

Type	# Elements	% in UIP Data
TextView	10,582	14.86%
Customized	5,075	7.13%
Spinner	1,962	2.75%
Others	784	1.10%
Total	18,403	25.84%

Table 7: Types of UIP Elements Other than EditText

data of classifier is not absolutely randomly selected (part of them are labeled by sensitive attributes automatically), a manual validation is required to confirm that the identification results of the classifier carries over the entire dataset. Second, the classifier is only capable of distinguishing UIP data elements from their textual semantics, the manual validation can be used to check whether static text labels are effectively excluded by UIPicker after behaviour based result filtering.

Evaluation of Classifier. The training set contains 53,094 elements in total, which includes 24,962 labeled by sensitive attribute values and financial-related elements, with 25,331 negative samples labeled by manual efforts.

We use ten-fold cross validation which is the standard approach for evaluating machine-learning classifiers. We randomly partition the entire set of sample elements into 10 subsets, and we train the classifier on nine of them and then test the remaining 1 subset. The process is repeated on each subset for 10 times. In the end, the average precision and recall is 92.5% and 85.43% respectively.

As shown in Table 8, we also compare the average precision and recall with other two classifiers, i.e., One-Class Support Vector Machine learning (OC-SVM) [32] and Naive Bayes [30]. The results show that the standard SVM performs the best. We tried OC-SVM with only positive samples (elements containing sensitive attributes) to train the classifier. OC-SVM generated more false negatives than the standard SVM due to the lack of negative samples. Naive Bayes, a traditional probabilistic learning algorithm, also produced very imprecise results. This happens especially when it deals with ele-

ments that contain low-frequency privacy-related texts.

Classifier	Avg.Precision	Avg.Recall
SVM	92.50%	85.43%
OC-SVM	93.74%	68.48%
Naive Bayes	95.42%	26.70%

Table 8: Classifier Comparison

Manual Validation. We envision UIPicker to be used as an automated approach for labeling elements that contain UIP data. UIPicker achieves this by using some easily available UIP data (elements containing sensitive attributes or hand-annotated) and then using the classifier to automatically explore larger parts of UIP data. Measuring precision is hard in this setting as there is no entire pre-annotated elements (labeling sensitive or insensitive for all of them) for a set of apps that could compare with UIPicker's identification results.

As a best-effort solution, we randomly select 200 apps from top 10 categories (20 in each) ordered by %apps which UIP data appear most in Table 4 as the manual validation dataset. As such categories may contain much more UIP data than others, it provides the opportunity that our experts can walk through less apps (and activities) to validate more UIP elements. The selected apps are excluded from the classifier's training process to avoid overlap. Such way can greatly improve the effectiveness of the manual validation. Since the subset of apps is randomly picked, we believe that the evaluation results can provide a reasonable accuracy estimation on the entire dataset. For each element that UIPicker identifies as UIP data, we check their corresponding descriptions in XML layout files with some automated python scripts for efficiency (quickly locating the element in layout files and trying to understand it from descriptions). If this is still insufficient for us to identify whether it is a UIP data element, we confirm them by launching the app and find the element in the layout screen. The manual validation over 200 apps shows that UIPicker identifies 975 UIP data elements with 67 false positives and 107 false negatives.

False Positives: The false positive rate is 6.4% (67/1042 elements UIPicker identifies). In most cases, this is caused by the element's neighbors. That is, the element's neighbors contain privacy-related texts while the element itself is not privacy-related. Consider the following example, an *EditText* with only one description "message" while its previous element requires the user to input username with many sensitive textual phrases. As UIPicker takes neighbor elements' texts into consideration for better identification results, the privacy-related texts in its neighbor make UIPicker falsely identify the current element as UIP data. We consider such false alarm as acceptable because once such false alarm happens, their neighbor elements (the actual UIP data elements) are very possible to be identified by UIPicker as well.

False Negatives: We manually inspect each app in the evaluation dataset by traversing their UI screens as much as possible to see whether there exists UIP data elements that missed by UIPicker. In 200 apps, we find 107 elements not identified by UIPicker as privacy-related, and we conclude the reasons as follows: (1) Some very low-frequency texts representing UIP were not inferred from UIPicker by the privacy-related text analysis module. For example, "CVV" represents the credit card's security code, however we find this only happened in 4 Chinese apps. The low occurrence frequency of texts like "CVV" in our croups makes UIPicker fail to add them as features for the identification process. (2) In static analysis for behavior-based element filtering, due to FlowDroid's limitations, the call trace of some element was broken in inter-procedural analysis which makes UIPicker miss such elements in the final output.

Based on the total number of TPs, FPs and FNs (975, 67, 107), we compute the precision and recall of UIPicker as follows:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$

Overall, UIPicker precisely identified most of UIP data, with 93.6% precision and 90.1% recall.

7.4 Runtime Enhancement Evaluation

System Overhead. We compare the performance overhead with TaintDroid using Antutu Benchmark [3]. We run Antutu 10 times in both systems under a Nexus Prime device, and the average scores are basically the same. This is reasonable because our mechanism only provides additional UIP data sources. We conclude that the security enhancement mechanism does not introduce noticeable additional performance overhead to TaintDroid.

Case Study. We find that some critical UIP data are under threats in Android apps. In Figure 7, a popular travel app "Qunar", which has 37 million downloads in China [12], sends users' credit card information with vulnerable SSL implementation during the payment process. The insecure transmission is reported to the user with a pop-up window when such data leave the device, thus the user can decide whether to proceed or use an alternative payment method to avoid the security risk.
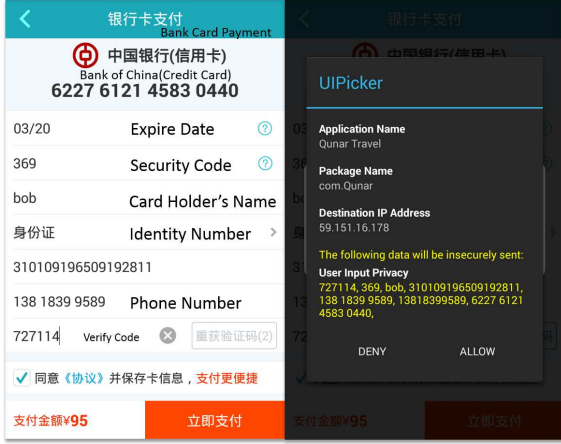
13

Figure 7: Insecure Transmission of UIP data. We use faked sensitive data in the experiment.

8 Discussion

In this section, we discuss the general applicability of UIPicker, as well as limitations and future work.

UIPicker is able to efficiently handle UIP data which previous work does not concentrate on, nor be able to cover. Compared with existing approaches that focus on System-Centric Privacy data, UIPicker rethinks privacy from a new perspective: sensitive data generated from user inputs, which is largely neglected for a long period. UIPicker provides an opportunity for users to make informed decisions in a timely manner when sensitive data leave the device insecurely, instead of letting users assume the app can be trusted.

UIPicker uses not only texts in UI screens but also texts in layout descriptions for UIP data identification. This framework is generic to all kinds of apps without locality limitation. The way UIPicker correlates UIP data from layout descriptions could also be leveraged by existing work [37, 28] that attempts to map the permission usage with app descriptions.

UIPicker has the following limitations. (1) UIPicker does not consider dynamically generated UI elements, although we have not found any UIP data element being generated at runtime in our experiments. Dynamic UI elements could be analyzed through more sophisticated static/dynamic analysis with the app's program code, which is our future work. (2) Currently, UIPicker can not handle sensitive user inputs in *Webview* because they are not included in app layout resources. In the future, we plan to download such webpages by extracting their URLs from the app, then analyze their text contents as well.

9 RELATED WORK

Privacy source identification. Existing work [16, 29] focuses on mapping Android system permissions with API calls. PScout [16] proposes a version-independent analysis tool for complete permission-to-API mapping through static analysis. SUSI [29] uses a machine learning approach to classify and categorize more Android sources and sinks which are missed by previous info-flow taint tracking systems. The most similar work with UIPicker is SUPOR [25], which also aims to automatically identify sensitive user inputs using UI rendering, geometrical layout analysis and NLP techniques. SU-PER mainly focuses on specific type of UI elements (EditText) while UIPicker is not limited to this.

Text analysis in Android app. Several studies utilize UI text analysis for different security proposes. As-Droid [24] detects stealthy behaviors in Android app by UI textual semantics and program behavior contradiction. However, it only uses a few keywords to cover sensitive operations such as "send sms", "call phone". CHABADA [21] checks application behaviors against application descriptions. It groups apps that are similar with each other according to their text descriptions. The machine learning classifier OC-SVM is used in CHABADA to identify apps whose used APIs differ from the common use of the APIs within the same group. Whyper [37] uses natural language processing (NLP) techniques to identify sentences that describe the need for a given permission in the app description. It uses Stanford Parser to extract short phrases and dependency relation characters from app descriptions and API documents related to permissions. AutoCog [28] improves Whyper's precision and coverage through a learning-based algorithm to relate descriptions with permissions. UIPicker could potentially leverage their techniques to generate more complete privacy-related texts for UIP data identification.

Static analysis. There are lots of work [24, 26, 15, 20, 34] on using static analysis to detect privacy leakage, malware or vulnerabilities in Android apps. AsDroid takes control flow graphs and call graphs to search intent from API call sites to top level functions (Activities). UIPicker's behavior-based result filtering is similar to AsDroid while they have different goals. SMV-HUNTER [34] uses static analysis to detect possible MITM vulnerabilities in large scale. The static analysis extracts input information from layout files and identifies vulnerable entry points from the application program code, which can be used to guide dynamic testing for triggering the vulnerable code.

14

10   CONCLUSION

In this paper, we propose UIPicker, a novel framework for identifying UIP data in large scale based on a novel combination of natural language processing, machine learning and program analysis techniques. UIPicker takes layout resources and program code to train a precise model for UIP data identification, which overcomes existing challenges with both good precision and coverage. With the sensitive elements identified by UIPicker, we also propose a runtime security enhancement mechanism to monitoring their sensitive inputs and provide warnings when such data insecurely leave the device. Our evaluation shows that UIPicker achieves 93.6% precision and 90.1% recall with manual validation on 200 popular apps. Our measurement in 17,425 top free apps shows that UIP data are largely distributed in market apps and our run-time monitoring mechanism based on UIPicker can effectively help user to protect such data.

References

[1] Amazon online store. https://goo.gl/jYdVPr.

[2] Android-apktool. https://goo.gl/UgmPXp.

[3] Antutu benchmark. https://goo.gl/78W9xL.

[4] Av-comparatives : Mobile security review - september 2014. http://goo.gl/JfmcYh.

[5] Bank app users warned over android security. http://goo.gl/PWcqUy.

[6] Cm security : A peek into 2014's mobile security. http://goo.gl/i58ihW.

[7] Google bouncer. http://goo.gl/ET4JDW.

[8] Monkeyrunner. http://goo.gl/AQsIQu.

[9] Natural language toolkit. http://goo.gl/qzWulA.

[10] Phishing attack replaces android banking apps with malware. http://goo.gl/cJqqyX.

[11] Python implementations of various stemming algorithms. https://goo.gl/kdxkqv.

[12] Qunaer 7.3.8. http://goo.gl/1vB2k7.

[13] scikit-learn. http://goo.gl/mBzGUZ.

[14] Wordnet, a lexical database for english. http://goo.gl/KwzO0r.

[15] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, p. 29.

[16] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 217–228.

[17] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Towards taming privilege-escalation attacks on android. In *NDSS* (2012).

[18] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *Proc. 23rd USENIX Security Symposium (SEC14), USENIX Association* (2014).

[19] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. vol. 57, ACM, pp. 99–106.

[20] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why eve and mallory love android: an analysis of android ssl (in)security. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer & Communications Security(CCS)* (2012).

[21] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *ICSE* (2014), pp. 1025–1035.

[22] HEUSER, S., NADKARNI, A., ENCK, W., AND SADEGHI, A.-R. Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium (SEC14)* (2014).

[23] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These arent the droids youre looking for. *Retrofitting Android to Protect Data from Imperious Applications. In: CCS* (2011).

[24] HUANG, J., ZHANG, X., TAN, L., WANG, P., AND LIANG, B. Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *ICSE* (2014), pp. 1036–1046.

[25] JIANJUN HUANG, PURDUE UNIVERSITY; ZHICHUN LI, X. X., AND WU, Z. Supor: Precise and scalable sensitive user input detection for android apps. In *Proc. of 24rd USENIX Security Symposium* (2015).

[26] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 229–240.

[27] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (2010), ACM, pp. 328–332.

[28] QU, Z., RASTOGI, V., ZHANG, X., CHEN, Y., ZHU, T., AND CHEN, Z. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1354–1365.

[29] RASTHOFER, S., ARZT, S., AND BODDEN, E. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)* (2014).

[30] RISH, I. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence* (2001), vol. 3, IBM New York, pp. 41–46.

[31] SALTON, G., WONG, A., AND YANG, C.-S. A vector space model for automatic indexing. *Communications of the ACM 18*, 11 (1975), 613–620.

[32] SCHÖLKOPF, B., PLATT, J. C., SHAWE-TAYLOR, J., SMOLA, A. J., AND WILLIAMSON, R. C. Estimating the support of a high-dimensional distribution. *Neural computation 13*, 7 (2001), 1443–1471.

[33] SMALLEY, S., AND CRAIG, R. Security enhanced (se) android: Bringing flexible mac to android. In *The 20th Annual Network and Distributed System Security (NDSS)* (2013).

[34] SOUNTHIRARAJ, D., SAHS, J., GREENWOOD, G., LIN, Z., AND KHAN, L. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 19th Network and Distributed System Security Symposium. San Diego, California, USA* (2014).

[35] SPYNS, P. Natural language processing. *Methods of information in medicine 35*, 4 (1996), 285–301.

[36] WEI, F., ROY, S., OU, X., ET AL. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1329–1341.

[37] XU, R., SADI, H., AND ANDERSON, R. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium* (2013), pp. 539–552.

[38] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium* (2012), pp. 539–552.

[39] YANG, Y., AND PEDERSEN, J. O. A comparative study on feature selection in text categorization. In *ICML* (1997), vol. 97, pp. 412–420.

[40] YANG, Z., YANG, M., ZHANG, Y., GU, G., NING, P., AND WANG, X. S. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1043–1054.

[41] ZHANG, Y., YANG, M., XU, B., YANG, Z., GU, G., NING, P., WANG, X. S., AND ZANG, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 611–622.

[42] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in android applications. In *The 20th Annual Network and Distributed System Security (NDSS)* (2013).

[43] ZHOU, Y., SINGH, K., AND JIANG, X. Owner-centric protection of unstructured data on smartphones. In *Trust and Trustworthy Computing* (2014), Springer, pp. 55–73.