This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2631949, IEEE Transactions on Information Forensics and Security

Identifying User-Input Privacy in Mobile Applications at a Large Scale

Yuhong Nan, Zhemin Yang, Min Yang, Shunfan Zhou, Yuan Zhang, Guofei Gu, Xiaofeng Wang, Limin Sun

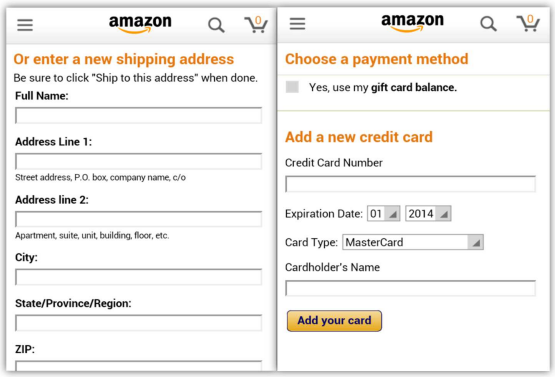
***Abstract*—Identifying sensitive user inputs is a prerequisite for privacy protection in mobile applications. When it comes to today's program analysis systems, however, only those data that go through well-defined system APIs (system controlled resources) can be automatically labelled. In our research, 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 13,072 top apps from Google Play, and find that 38.69% of them involve sensitive user inputs. Just like system controlled resources, these data are also exposed to a series of privacy leakage threats. For these sensitive user inputs, manually marking them involves a lot of efforts, impeding a large-scale, automated analysis of apps to defend against potential privacy leakage.**

To address this important issue, we present *UIPicker*, an adaptable framework for automatic identification of sensitive user inputs as the first step. 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 evaluate our approach over — randomly selected popular apps on Google-Play. UIPicker is able to accurately label sensitive user inputs most of the time, with 94.0% precision and 96.0% recall.

***Index Terms*—Android security, privacy protection, user input privacy, privacy leakage.**

I. Introduction

PROTECTING user's sensitive data within mobile applications (*apps* for short) has always been at the spotlight of mobile security research. Lots of program analysis techniques have been developed, or applied to evaluate potential information leakage for mobile apps, either dynamically [1], [2], [3] or statically [4], [5], or in a hybrid way [6]. Besides, access control mechanisms [7], [8], [9], [10] have also been proposed to enforce fine-grained security policies on the way that private user data can be handled on a mobile system.



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

Figure 1. Examples of User-Input Privacy (UIP) Data. (a) requires user to input his/her detailed address for delivering products. (b) requires user to input the credit card credential to accomplish the payment process.

The complete labelling of sensitive user data is critical to those privacy protection mechanisms. Some of the privacy 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 user privacy is the content the user enters to a mobile app through its user interface (UI), such as credit-card information, username, password, etc. Figure 1 shows two UI screens that contain sensitive user inputs in the Amazon Online Store [11] app. 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 leakage threats. It has been reported [12], [13], [14], [15] 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

This work was supported in part by the National Program on Key Basic Research under Grant 2015CB358800, in part by the Science and Technology Commission of Shanghai Municipality under Grant 15511103003, and Grant 13JC1400800, and in part by the National Natural Science Foundation of China under Grant 61602121, Grant 61602123, and Grant 61300027, and in part by the open project of Beijing Key Laboratory of IOT information security technology under Grant J6V0011104. (Corresponding authors: Zhemin Yang, and Min Yang.)

Y. Nan, Z. Yang, M. Yang, S. Zhou, and Y. Zhang are with the School of Computer Science, Fudan University, Shanghai 201203 China (email:nanyuhong, yangzhemin, m_yang, sfzhou11, yuanxzhang@fudan.edu.cn).

G. Gu is with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77840 USA (email: guofei@cse.tamu.edu).

X. Wang is with the Center for Security Informatics, Indiana University at Bloomington, Bloomington, IN 47405 USA (email: xw7@indiana.edu).

L. Sun is with the Beijing Key Laboratory of IOT information security technology, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093 China (email: sunlimin@iie.ac.cn).

eavesdropping attacks. Recent work further shows that side channels [16] and content-pollution vulnerabilities [17] can be leveraged to steal sensitive user inputs as well. In our research, we found that among 13,072 top Google-Play apps, 38.69% require users to enter their confidential information.

Although the UIP data urgently needs protection, the big difference with system-controlled user data makes its technical solution by no means trivial. Unlike system-controlled user data, which can be easily identified from a few API functions, the UIP data cannot be found without interpreting the context and semantics of UIs. A straightforward approach is to mark all the inputs as sensitive [4], which is clearly an overkill and will cause a large number of false positives. Prior approaches [18], [4], [6] 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.

To address this issue, we therefore propose a novel framework called UIPicker for the 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 irrelevant elements 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.

We implemented UIPicker based on python NLTK module [19] with DroidSafe [20], and built our identification model using 13,072 popular Google Play apps. Our evaluation of UIPicker over 500 randomly selected popular apps shows that it achieves a high precision (94.0%) and recall (96.0%).

The identification results based on classified top free apps show that in some app categories, more than half of apps contain UIP data. With the help of UIPicker, we also conduct a quantitative measurement about the UIP data distribution in different datasets collected in different times, which helps us understand how the existence of UIP data changes with apps' quick evolvement. The comparative results between different datasets ranging from 2012 to 2014 shows that the amount of UIP data increases quickly among app-markets. Further protection of these UIP data is in urgent need.

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 also be easily deployed to support any existing static and dynamic taint analysis tools as well as access control frameworks for automatic labelling of private user information. Although the prototype of UIPicker is implemented for Android, the idea can be applied to other platforms as well.

Contributions. We outline the paper's contributions below:

- New Understanding: We raise the issue that compared with system-controlled data, UIP data are equally important and urgently needs protection. We show the numerous existences of UIP data in real-world app markets across all categories. We also measure the distribution of UIP data with multiple datasets collected in different periods, which fully validates its importance in practice and helps us gain insights.
- New techniques: We propose UIPicker, a series of techniques for automatically identifying UIP data. We use NLP techniques to automatically cluster privacy-related texts from a corpus of android layout resources, and combine machine-learning with program analysis techniques to identify sensitive user inputs at a large scale. Lots of existing tools can benefit from UIPicker for better privacy recognition in mobile applications.
- Design, implementation, and evaluation: We conduct a series of evaluations to show the effectiveness and precision of UIPicker. The prototype of UIPicker shows that compared with other alternative approaches, the techniques introduced in the paper achieve good results in practice, and also are highly extensible.

This work is an extension of the conference version appearing in the Proceedings of the 24th USENIX Security Symposium [21]. Compared with the conference version of the paper, this manuscript employs a new UI classification mechanism for privacy-related text analysis to avoid the required human intervention for removing noisy texts, and performs a new experiment to evaluate its effectiveness. This manuscript also adopts other changes in different steps to reduce the false negatives of the framework. This manuscript presents a more clear description about the identification approach, as well as a more detailed evaluation and experiments. Besides, this manuscript introduces a new trend analysis about UIP data distribution variations between different datasets ranging from 2012 to 2014.

The rest of this paper is organized as follows. Section II gives the motivation and challenges for identifying UIP data, then introduces background knowledge about Android layout. Section III gives an overview of UIPicker and illustrates the key techniques. Section IV describes the identification approach step by step. Section V gives some implementation details and Section VI gives evaluation about the framework. Section VII details its application scenarios and Section VIII discusses the limitations. Section IX and X describes related work and concludes our work.

II. Motivation

In this section, we first provide a motivating example of users' sensitive input in two UI screens and security implications about UIP data, then we investigate challenges in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2631949, IEEE Transactions on Information Forensics and Security

identifying them. We also give some background knowledge about Android layout resources for further usage.

A. Security implications of UIP data

Most of the UIP data are personal information that users are unwilling to expose to public, or any 3rd parties. However, various threats to the UIP data exist in the real world. Here we summarize the security implications as follows.

Unintended exposures. For benign apps, many of them require users to input sensitive information for providing customized functionalities, but failed to keep their security. Existing report [14] showed that less security-savvy developers transmit user data in plain text, which indeed include sensitive user inputs. Furthermore, previous work [22] showed that a large number of apps implement SSL with inadequate validations (e.g., apps contain 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 like banking accounts, passwords, etc.

Deliberate harvesting. Malware can steal sensitive user inputs by exploiting the weakness inside existing protection mechanisms. For example, fraud apps mimic the UIs of popular apps for seducing users to input their sensitive data [12], resulting heavy financial losses. Besides, it is quite noticeable that ad-plugins are also capable of reading user inputs [23] because they share the same privilege with the apps it hosted. Recent work [24] [25] showed that malicious ad libraries can infer sensitive information such as users' search history about pharmacy. The significant correlation between ads and user profiles reveals that 3rd-parties are very interested in sensitive user inputs such as search history.

B. Challenges

Given the situation that 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, then we find 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 [4] 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 when it comes to runtime protection about 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.

UIP data can be easily recognized by human. However, it is quite challenging for the machine to automatically identify with existing approaches in large-scale. For runtime monitoring, unlike the highly structured system-controlled data, sensitive user inputs can not simply matched by regex expressions when users fill them into the screen. Besides, like any normal inputs, privacy-related inputs are sparsely distributed in various layouts in a single app, touching such data may require login or complex trigger conditions. It is very difficult for automatic testing tools like [26], [22] 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.

In this work, we identify UIP data 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.

C. Android Layout Background

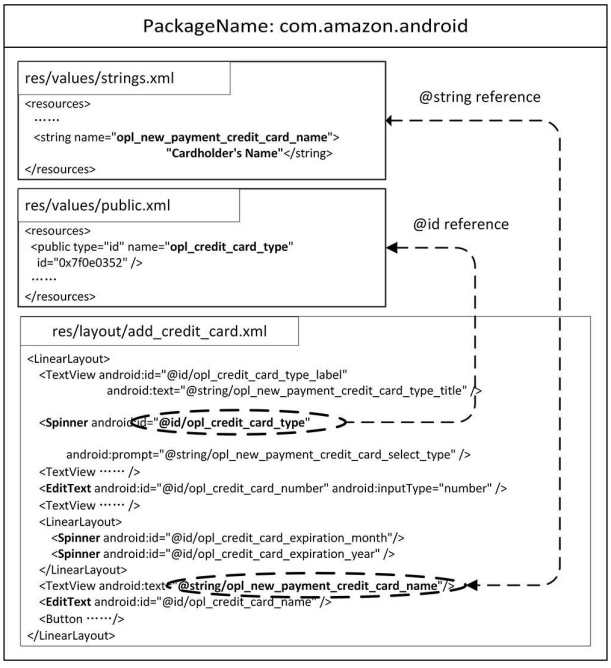


Figure 2. A sample of Android layout description resources

In Android, a Layout (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 to provide additional information about the element.

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_name*, which means the content showed in this label will be the referenced string "Cardholderś Name" in */res/values/stings.xml*.

III. System Overview

In this section, We first clarify our identification scope of UIP data, then we give an overview of UIPicker as well as the key techniques applied in our framework.

A. Identification Scope

UIP data could be any piece of data that users or app analysts consider to be sensitive from inputs. In the current version of UIPicker, we consider the following 3 categories:

- **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.

There are other various sensitive inputs like search history, vehicle identification number, or any personal data that closely related to user's privacy. In this work, we limit the identification scope to those 3 categories because they cover most existing UIP data in current apps, and are easy for statistical illustration. As further discussed in Section IV-B, by assigning the type of UIP data, UIPicker can be easily extended to identify other types of sensitive user inputs. Also 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.

B. Overall Architecture

As Figure 3 shows, 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-Processing 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. Leveraging an English dictionary like WordNet [27] 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 they can not be correlated from a dictionary like WordNet.

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 from a given set of layout resources. 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 [28] 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, the phrase "address such problem" showed in a layout screen 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.

Behaviour Based Result Filtering. Besides identifying el-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2631949, IEEE Transactions on Information Forensics and Security

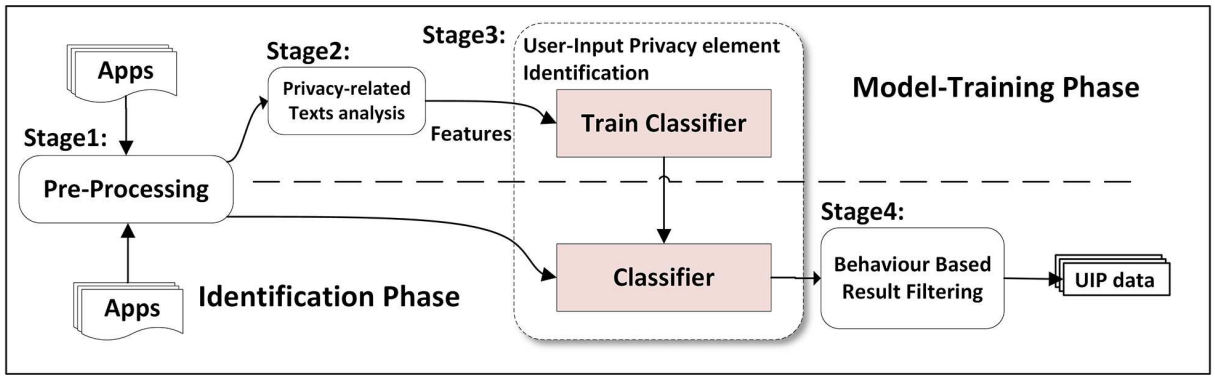


Figure 3. System overview of UIPicker

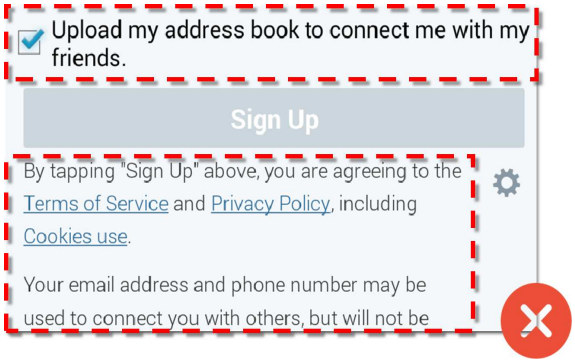


Figure 4. Examples of static elements which need to be filtered out

ements 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 as showed in figure 4. Although Android defines *EditText* 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 behaviour, thus filter out irrelevant elements we do not expect.

IV. IDENTIFICATION APPROACH

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

A. Stage 1: Pre-Processing

Resource Extraction. We first decode the Android APK package with apktool [29] 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.

- **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]*. Besides, we also extract constant strings in app code which are displayed in the UI dynamically. They are commonly assigned as the parameters of method *UIObject.setText*.
- **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.

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 I.

Table I
SELECTED LAYOUT RESOURCE SAMPLES

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

Given the UI-related content, we first split words connected with delimiters or CamelCase like PhoneNumber. Then, we remove Non-English strings, non-text characters like digits, punctuation, and stop words, because these words do not carry any sensitive information to users. After that, we use Stemming described in [30], to transform inflected (or sometimes derived) words to their stems. For example, we change the words like secured, security into the uniformed format secure. All these steps can greatly improve the results of later

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2631949, IEEE Transactions on Information Forensics and Security

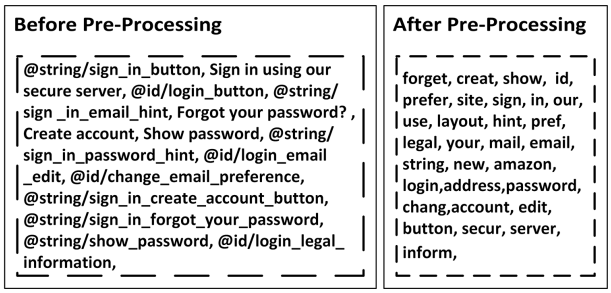


Figure 5. Text samples with Pre-Processing

identification processes since they reduce the number of words and their variants in the selected resources.

Figure 5 shows part of texts before and after pre-processing 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".

B. Stage 2: Privacy-related Texts Analysis

In this stage, we use Chi-Square test [31] to extract privacy-related texts from a subset of specific UI layouts. The intuition here is that privacy-related terms prefer to be correlated in specific UI 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 test is a statistical test that 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 UI or non privacy-related UI).

Here we choose *UI texts* rather than *layout descriptions* to generate privacy-related texts due to the following reasons. First, *layout descriptions* are not well structured as the naming behaviours 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 6 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. Here we use heuristic rules to automatically classify layout samples. Table II shows what we used as initial seeds for inferring privacy-related texts for privacy categories that defined in Section III-A. These seeds are easy to come up with by human instinct regarding to different privacy categories. As showed in the last row of the table, one can also easily configure other topic like "Health" to expand the identification scope of UIPicker.

Table II
INITIAL SEEDS USED FOR PRIVACY-RELATED TEXTS CLUSTERING

Type	Initial Seeds
Account Info	user name, password, email, login
Location	address, zip code, city
Financial	credit, security, expire, card
Health	weight, height, blood

Heuristic-based automatic UI classification. Our heuristic rules for UI classification come from the observation that when a UI layout contains several user sensitive keywords, it is likely that the other keywords on the same page are also privacy-related. Given a UI layout, we first scan its text resources with the text seeds. The more privacy-related texts hit in a given UI, the more it is likely to be privacy-related. Thus, for those layout samples whose scan hits are no less than a pre-defined threshold, we label them as privacy-related (positive samples). Because the number of seeds varies from 3 to 5, our current setting of the threshold does not exceed 4. Especially, since we only have 3 seeds for the category "Health", the threshold on type Health is also set to 3. On the other hand, for layout samples that do not contain any of texts appeared in the initial seeds, we label them as privacy-irrelevant (negative samples). Note that we do not label those layouts containing only a single or two 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.

Alternative Approaches. Other alternatives like Bag-of-Words and Skip-Gram architectures are often used for word-clustering for a given topic as well. In our research, we compared our Chi-Square based text clustering approach with these two popular alternatives. However, both of them are are found to be less effective due to our specific scenarios, as we further discuss in Section VI-B.

C. Stage 3: UIP Data Element Identification

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2631949, IEEE Transactions on Information Forensics and Security

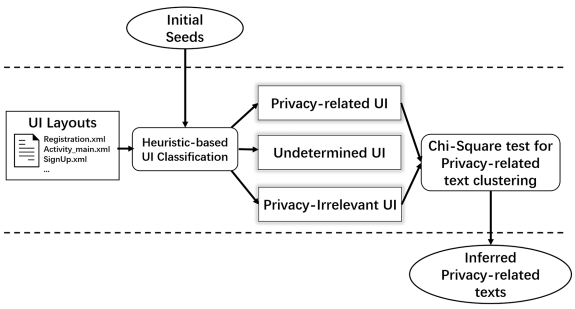


Figure 6. Privacy-related text clustering

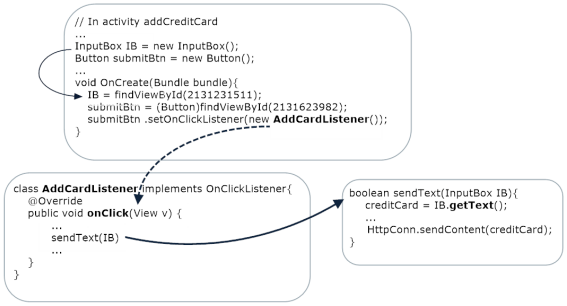


Figure 7. Sample codes for requesting a user's credit card number

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. 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 on top of them.

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. The training data is automatically constructed as follows: First, we label all elements with sensitive attributes[1], shown in table III as positive samples since they are obviously a subset of UIP data elements which Android defines. For other types of privacy-related elements(e.g., Financial) that are not covered by any sensitive attributes that Android provides, we leverage the two classified layout sample sets in Stage 2 to automatically generate the training data. Specifically, for any input element from privacy-related layout samples, if it contains any keywords inside the initial seeds, we label it as a positive sample. On the other hand, for any input element which comes from privacy-irrelevant layout samples and does not contain any of the initial seeds, it is selected as a negative sample.

Table III
SENSITIVE ATTRIBUTE VALUES IN LAYOUT DESCRIPTIONS

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

[1]In some older apps, developers also use specific attributes like "android:password=True" to achieve the same goal as *inputType*. We list them in Table III and call them sensitive attribute values as well for simplicity.

Classifier Selection. We utilize the standard support vector machine (SVM) as our classifier. SVM is widely used for classification and regression analysis. 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.

D. Stage 4: Behaviour 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.

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 7 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.

V. IMPLEMENTATION

Dataset. Two datasets are used as the training set and evaluation set respectively. For the training set, we crawled top 500 apps from Google Play Store based on its pre-classified 35 categories in Oct. 2014, that is, 17,425 apps in total. For the evaluation dataset, we crawled another 13,072 apps collected from Google Play in Dec. 2015, which are top 540 apps among

25 app categories. Since 11 app categories are merged into a single category (Game) during our second crawling, our second dataset is smaller than the first one.

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) based on the NLTK [19] module. The last step, static analysis for result filtering, is implemented in Java, which extends DroidSafe [20] and introduces additional 1020 LOCs. All experiments are performed on a 32 core CentOS server with Linux 3.10 kernel and 192GB memory.

For privacy-related text analysis, the initial seeds for each privacy category are assigned as texts in Table IV. For each privacy category, we conduct the privacy-related UI classification, and Chi-Square test using apps in the training 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 privacy category, we collect the top 100 words from its Chi-Square test results. Over all, UIPicker extracts 245 privacy-related terms from 13,308 distinct words (some texts may overlapped in different privacy categories). We list part of them in Table IV corresponding to the privacy category they belong to. Such data are used as features for privacy-related element identification in the follow-up step.

Table IV
PART OF INFERRED PRIVACY-RELATED TEXTS FROM CHI-SQUARE TEST

Privacy Category	Representative Inferred Texts(Stemmed)
Login Credenticals &User Profile	mobil phone middl profile cellphon account nicknam firstnam lastnam person birth login confirm detail regist
Location	provinc street postal locat countri home
Financial	secur month date pay year bill expir debit transact mm yy pin code

The SVM classifier is implemented with scikit-learn [32] in poly kernel. We optimize the classifier parameters (gamma=50 and degree=3) for performing the best results.

For each element identified as privacy-related by the SVM classifier, UIPicker conducts static taint analysis to check whether it satisfies specific behavior described in Section IV-D. Since DroidSafe successfully handles android life cycle (system event based callbacks) and UI widgets, the call graph and invocation context for each method should be both precise and complete. We get each element's def-use 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.

VI. EVALUATION

In this section, we present our evaluation results. We first show the performance of UIPicker in Section VI-A, and evaluate the key technique applied in UIPicker, the privacy-related text clustering approach, in Section VI-B. Then, we discuss the effectiveness and precision of UIPicker in Section VI-C and Section VI-D. We compare our approach with a relevant work, SUPOR, in Section VI-E. Then a trend analysis about UIP data distribution is given in Section VI-F, which shows that the number of apps containing UIP data increases quickly, thus the protection of such data is in urgent need.

A. Performance

During our experiment, the training phase of the classifier takes 2.5 hours, and the identification phase takes 23.5 hours (6.47 seconds per app). Pre-processing time for apps is included in both phases. Among the overhead of our approach, behaviour based result filtering (Stage 4) is the most time-consuming one, while other parts of our analysis (including pre-processing, privacy-related texts analysis and UIP data element identification) take only 24% of the total time. Since UIPicker mainly targets for customized system vendors or security analysts, we consider such overhead quite acceptable.

B. Privacy-related texts clustering

To reveal which method is more suitable for privacy-related text clustering, we compare the Chi-Square test algorithm with two other possible alternatives: Bag-of-Words and Skip-Gram. The results show that the Chi-square performs best in our purpose.

The Bag-of-Words and Skip-Gram architectures are often used for word-clustering for a given topic as well. Both of them construct a set of vocabularies from the given set of training text data and then learn vector representations of words. The results can be used as features in many natural language processing and machine learning applications.

To compare these two alternatives with our text clustering approach, we implement them using word2vec [33], and evaluate their abilities to extract privacy-related texts. During our evaluation, all privacy-related UIs, which are the same input set as our Chi-Square based approach, are used as corpus of the training set. As revealed in Table V, both Bag-of-Words and Skip-Gram approaches report lots of irrelevant keywords as the privacy-related texts, which dramatically decreases the precision for using them as UIP data identification vectors. By observing the analysis results, we find that these two alternatives perform badly with a sparse training set. Actually, instead of sentences or paragraphs which are commonly used in Bag-of-Words or Skip-Gram approaches, only a few words are presented in each user interface. Thus, the information is too sparse to be an effective training set for these approaches. Besides, using Chi-Square test, we can filter out irrelevant keywords using negative samples (privacy-irrelevant UI), while both Bag-of-Words and Skip-Gram only consider positive samples, thus their precision cannot be improved by utilizing the negative samples.

C. Effectiveness

UIP Data Distribution. We show the general identification results of UIPicker in Table VI. In 13.072 apps, UIPicker finds that 5,307 (38.69%) contain UIP data. We list our results in

Table V
ALTERNATIVE APPROACHES FOR PRIVACY-RELATED TEXTS CLUSTERING.

Top inferred texts	% of irrelevant keywords in average		
	Bag-of-Words	Skip-Gram	Chi-Square
25	27%	18%	9%
50	34%	25%	12%
100	45%	32%	13%
150	61%	59%	31%

a descending order of the identified total app amounts. As we can see, in some categories ranked in top of the list, nearly half of apps contain UIP data.

We make the following observations from this table. First, application categories such as Shopping, Business, Finance, Social and Communication are more likely to request Account Credentials and User Profile information, which shows that these apps are closely related to users' personal activities. It is also reasonable that apps in categories like Shopping, Business, Travel & Local require many financial-related sensitive inputs, because many of these apps provide payment functionality.

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 which we list in Table VII can be regulated by fixed system APIs. 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 VIII shows, in our dataset, 4,206 apps use system defined APIs for requesting Account Credentials and Profile Information while UIPicker identifies 3,613 (27.64%) apps containing UIP data. 1,644 (12.57%) apps request financial privacy data from users, and none of system defined APIs can regulate such data. In general, UIPicker identifies 5,307 (38.69%) apps containing at least one category of UIP data, which have been largely neglected by previous work in privacy security analysis and protection.

Note that, there is some overlap between system defined APIs and UIP data. 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 getLastKnownLocation() 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 a shopping address for delivering goods.

Comparing to sensitive attribute values. In Section IV-C, 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 VIII shows, in general, UIPicker identifies 66,382 more UIP data elements than simply identifying them by sensitive attribute values (e.g., textPassword). Especially for the Location category, UIPicker identifies 11,675 elements, which is nearly 14 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 IX. In general, UIPicker finds 17,088 (25.74%) elements other than *EditText* to accept users' sensitive inputs. Our approach successfully identifies more UIP data, compared with similar work SUPOR [34] which can only identify UIP data with some fixed type of input fileds like *EditText*. More details will be discussed in Section VI-E. 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. Type "Others" in table contains elements such as *RadioButton*, *CheckBox*.

D. Precision

For evaluating the precision of UIPicker, we perform the evaluation of classifier based on the machine-leaning dataset mentioned in Section IV. We also conduct a manual validation for two reasons. First, since the training data of classifier is not absolutely randomly selected (part of them are labelled 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 labelled by sensitive attribute values and financial-related elements, with 25,331 negative samples.

In order to evaluate the classifiers, we perform a nested 10-fold cross validation (CV for short) [35]. The nested CV is conducted with 2 phases: an inner CV loop which is used to perform the tuning of the parameters and an outer CV to compute an estimate of the error. Thus, we first randomly partition the entire dataset into 10 subsets, use 1 subset as the outer CV test set, and use the remaining 9 subsets in the inner CV. When tuning parameters during inner CV, we re-partition the dataset (which is 9/10th of the original dataset) into 10 subsets, and conduct 10-fold cross validation. Both the inner and outer CV process is repeated 10 times over different test

Table VI
UIP DATA DISTRIBUTION AMONG 25 CATEGORIES.

Application Category	Account Credentials & User Profile		Location		Financial		Total	
	#element	%app	#element	%app	#element	%app	#element	%app
Shopping	5,394	67.37%	1,943	53.36%	893	32.05%	8,230	68.00%
Business	4,761	50.39%	930	53.54%	362	20.08%	6,053	57.05%
Finance	5,912	43.99%	664	48.06%	1,355	35.27%	7,930	55.59%
Social	3,329	39.84%	546	47.01%	270	17.33%	4,145	50.68%
Communication	2,544	36.97%	461	39.60%	204	16.36%	3,209	49.98%
News & Magazines	1,448	28.71%	314	39.06%	149	11.91%	1,910	49.34%
Travel & Local	3,749	33.83%	1,089	37.13%	700	22.40%	5,537	46.94%
Productivity	1,851	28.93%	393	39.26%	121	9.92%	2,366	46.72%
Transportation	1,943	43.51%	636	35.31%	448	18.51%	3,027	46.38%
Media & Video	1,909	22.91%	246	31.48%	155	13.06%	2,311	45.31%
Tools	1,960	29.75%	377	29.55%	140	10.57%	2,477	44.62%
Weather	789	35.98%	561	18.98%	92	10.82%	1,443	40.92%
Lifestyle	2,503	34.73%	662	30.34%	215	12.40%	3,380	40.08%
Sports	1,466	26.11%	343	26.31%	138	10.83%	1,947	38.72%
Entertainment	1,483	17.68%	214	22.95%	109	7.79%	1,806	32.67%
Medical	1,025	20.96%	405	23.65%	140	7.69%	1,570	31.47%
Education	1,162	16.83%	118	22.50%	68	6.85%	1,348	29.46%
Health & Fitness	1,071	19.11%	267	19.32%	124	7.65%	1,462	28.15%
Libraries & Demo	1,184	15.42%	728	19.47%	313	12.58%	2,225	24.30%
Books & Reference	743	13.41%	321	16.37%	151	7.30%	1,214	23.63%
Music & Audio	693	11.73%	178	13.92%	66	4.97%	937	23.54%
Photography	645	12.47%	119	14.00%	35	2.63%	799	20.93%
Comics	335	8.51%	102	13.66%	29	2.18%	466	16.79%
Game	125	1.49%	10	6.18%	22	0.85%	156	16.32%
Personalization	360	8.35%	49	4.02%	23	2.16%	432	9.21%
Total	48,385	27.64%	11,675	28.99%	6,322	12.57%	66,381	38.69%

Table VII
SYSTEM DEFINED SENSITIVE APIS RELATED TO UIPICKER'S IDENTIFICATION SCOPE

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()

set respectively. As a result, the average precision and recall rate is 92.89% and 86.92% respectively.

As shown in Table X, we also compare the average precision and recall with other two classifiers, i.e., One-Class Support Vector Machine learning (OC-SVM) [36] and Naive Bayes [37]. The results show that the standard SVM performs the best. We tried OC-SVM with only elements with sensitive attributes to train the classifier. OC-SVM generated more false positives and 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 elements that contain low-frequency privacy-related texts.

Manual Validation. We envision UIPicker to be used as an automated approach for labelling elements that contain UIP data. UIPicker achieves this by using some easily available UIP data (elements containing sensitive attributes or auto-labelled by heuristic rules) 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 (labelling 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 500 apps from all categories (20 in each) in Table VI as the manual validation dataset. 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 500 apps shows that UIPicker identifies 769 UIP data elements among 1,603 input fields, with 49 false positives and 32 false negatives.

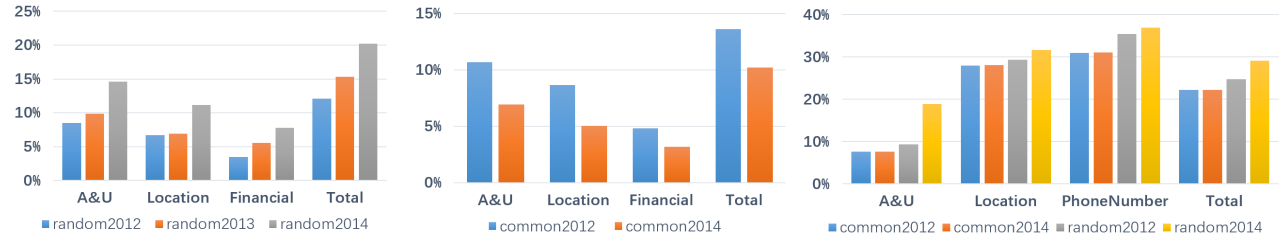
False Positives: The false positive rate is 6.0% (49/818 elements UIPicker identifies). In most cases, this is caused by the element's neighbours. That is, the element's neighbours contain privacy-related texts while the element itself is not privacy-related. Consider the following example, an *EditText* with only one description "message" while its previous element requires the user to input username with many sensitive textual phrases. We consider such false alarm as acceptable because once such false alarm happens, their neighbour 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 500 apps, we find 32 elements not identified by UIPicker as privacy-related, resulting a false negative rate of 4.0% (32/801) and we conclude the reasons

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2631949, IEEE Transactions on Information Forensics and Security

Table VIII
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).

Privacy Category	System Defined APIs (#Apps)			Elements with Sensitive Attribute Values (#Elements)		
	API	UIPicker	Overlap	InputType	UIPicker	Incremental
Account Credentials & User Profile	4,206	3,613	1,291	22,385	48,385	26,000
Location	11,205	3,790	2,960	831	11,675	10,844
Financial	-	1,644	-	-	6,322	-
Total	12,603	5,307	-	23,216	66,382	43,166



(a). UIP data distribution in random datasets (B). UIP data distribution in common datasets (c). Permission usage statistics in different datasets

Figure 8. Trend analysis about UIP data distribution

Table IX
TYPES OF UIP ELEMENTS OTHER THAN EDITTEXT

Type	# Elements	% in UIP Data
TextView	9,926	14.95%
Customized	4,707	7.09%
Spinner	1,802	2.71%
Others	653	0.98%
Total	17,088	25.74%

Table X
CLASSIFIER COMPARISON

Classifier	Avg.Precision	Avg.Recall
SVM	92.89%	86.92%
OC-SVM	72.01%	69.69%
Naive Bayes	96.33%	20.03%

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 corpus makes UIPicker fail to add them as features for the identification process. (2) In static analysis for behaviour-based element filtering, due to DroidSafe's limitations, the call trace of some element was broken in inter-procedural analysis in some very complicated apps, which makes UIPicker miss such elements in the final output. However, all such false negatives can be easily reduced by employing new techniques such as privacy-related text clustering and static analysis for application codes.

Based on the total number of TPs, FPs and FNs (769, 49, 32), we compute the precision and recall of UIPicker as follows:

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

Overall, UIPicker achieves 94.0% precision rate and 96.0% recall rate, showing that it successfully identified most of UIP data.

E. Comparison with SUPOR

In this section, we evaluate our method by comparing its results to SUPOR [34], which has a similar goal as UIPicker. Similar to UIPicker, SUPOR first selects a set of sensitive keywords. Then, leveraging the selected keywords, it launches a static analysis to identify which input fields are sensitive.

Compared to our methodology, SUPOR manually selects only 23 sensitive keywords for our selected 3 categories, which is much fewer than UIPicker (245 keywords). Due to the limited keywords, many sensitive user inputs described by other terms cannot be recognized by SUPOR. UIPicker employs a fully automatic approach to infer a detailed privacy related keywords, including the commonly used keywords and those which are rarely used but present the similar meanings. For example, for account registration, SUPOR uses *log in, sign in, register* as the keyword list. However, UIPicker infers that other texts, such as *sign up, create, and finish*, have the similar user intention. As a result, UIPicker identifies more UIP data compared with SUPOR.

Since SUPOR is a private property of NEC Labs, neither the source code nor the executable binary is publicly available. Thus we cannot compare it with UIPicker directly. As a best effort comparison, we implement a simplified UIPicker which only considers the sensitive keywords used in SUPOR, and evaluate its effectiveness on identifying sensitive UI elements.

Table XI
COMPARE WITH SUPOR. THE SIMPLIFIED UIPICKER USED ONLY SENSITIVE KEYWORDS IN SUPOR FOR COMPARISON

Element Type	Identified Elements by UIPicker			Identify Capabilities[1]	
	Simplified	Full	increment	SUPOR	UIPicker
EditText	36,684	49,294	34.37%	●	●
TextView	7,078	9,926	40.24%	○	●
Customized	3628	4,707	29.74%	○	●
Spinner	1,410	1,802	27.80%	○	●
Others	496	653	31.65%	○	●
Total	49,296	66,382	34.66%	-	-

[1] ●/○ denotes capable/unable to identify UIP data with this element type.

As showed in Table XI, using the keyword list provided by SUPOR, about 34.66% UIP data will be missed.

Besides, SUPOR relies on the UI layout descriptions (XML files) to understand the purpose of user inputs. Since many customized input fields are not explicitly labelled in layout resources, it is difficult to identify such inputs solely upon layout files. As showed in column 5-6 of Table XI, SUPOR only covers UI elements whose type (or super-type) is *EditText*. However, revealed by our evaluation results in Table IX, over 25.74% of sensitive inputs are not standard Android input fields (*EditText*). UIPicker detects whether a UI element is input field by analyzing its behavior reflected by the app's code, thus it can cover more input fields in apps.

F. Trend analysis about UIP data distribution

In order to learn how the distribution of UIP data evolved with apps in different time periods, we conduct a measurement study with additional datasets. These apps are selected as follows against those in top-free apps:

- **Random Datasets.** The random datasets consist of 3 sets of apps crawled from GooglePlay in different time ranging from 2012 to 2014, as we label them as *random-12, random-13 and random-14* separately. Each of the dataset contains 10,000 randomly selected apps. we consider such datasets can represent the overall app status in different years.
- **Common Datasets.** We also extract 14,923 apps that exist in both our datasets crawled in 2012 and 2014 based on their package names. Then, we remove the apps which have identical MD5 values because they may no longer be updated by the developers since 2012. Thus, these datasets contain the apps that exist in GooglePlay for two years and at least have been updated once. We label them as *common-12* and *common-14* separately.

As Figure 8(a),(c) showed, compared with permission-related privacy usage, the number of apps containing UIP data increases quickly, which reveals that the usage of UIP data will be more promising in the foreseeable future. Since more apps require UIP data, causing the risk of privacy leakage rises rapidly, the further protection of such data is in urgent need.

The *Common Datasets* reveal how the UIP data distribution changes in the same apps at different times. As shown in Figure 8(b), The amount of UIP data surprisingly drops for the apps with same package names. Our manual analysis demonstrates that this happens because many of these apps utilize third-party SDKs to handle user information, instead of processing it themselves. For example, the Single-Sign-On SDKs like Facebook Connect provides APIs to help developers import user profiles directly from their SSO accounts. Besides, the payment SDKs such as Alipay and Paypal can redirect the payment process into their corresponding payment apps instead of asking users to type their banking credentials in all apps. Overall, in the *Common-14* dataset compared with *Common-12*, 1,773 out of 14,923 (12%) apps introduce at least one kind of third-party SDKs which contain sensitive user data after their upgrade. However, this observation does not necessarily mean that the user's sensitive data will be secure, because they just come from different source other than direct user input and they still need security protection as their sensitive nature does not change.

VII. UIPICKER APPLICATION

Effective and efficient recognition of sensitive data sources is the key to various privacy leakage detection/prevention techniques. Since existing approaches cannot recognize user-input privacy, they are unable to detect/prevent the corresponding privacy leakage. Besides, since there are huge amounts of Android apps, it is impractical to label privacy sensitive data manually by users or security analysts. UIPicker proposes an automatic approach to identify those user-input sensitive data, which can enhance the state-of-the-art security analysis/enhancement tools in the following ways:

Privacy leakage detection. Privacy leakage detectors can integrate UIPicker to precisely detect disclosure of sensitive user inputs. By recognizing privacy related user inputs, and labeling these inputs as sensitive sources, we can analyze their possible propagation destinations with static data propagation analysis. Utilizing the analysis report, app developers can check whether the sensitive user inputs are transmitted out without proper verification, and app markets can report possible privacy leakages to users.

Runtime security enhancement. 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 popup window when such data insecurely leave the device, thus effectively mitigating the potential threats posed by apps.

As a detailed illustration of this scenario, we developed a prototype that integrates UIPicker with TaintDroid, and reports the following insecure app behavior:

- **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 [22] 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2631949, IEEE Transactions on Information Forensics and Security

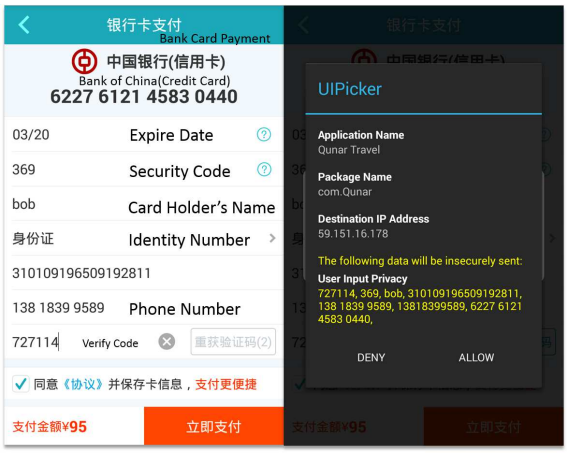


Figure 9. Runtime protection of UIP data

carry over critical sensitive data in most cases. UIP data should not be transmitted in this way as well.

As shown in Figure 9, a popular travel app "Qunar", which has 37 million downloads in China [38], 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.

VIII. Discussion and Limitation

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 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 [28], [39] 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.

IX. Related Work

Privacy Source Identification. Existing work [40], [41] focuses on mapping Android system permissions with API calls. PScout [40] proposes a version-independent analysis tool for complete permission-to-API mapping through static analysis. SUSI [41] 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 [34], which also aims to automatically identify sensitive user inputs using UI rendering, geometrical layout analysis and NLP techniques. SUPOR mainly focuses on specific type of UI elements (EditText) while UIPicker is not limited to this.

Sensitive User Inputs Protection. Cashtags [42] protects users' sensitive inputs by intercepting and replacing them with non-sensitive data when they are showed in layout screens, thus preventing data leakage based on visual observations. Chen et al. [43] propose I-BOX, a sandbox-based mechanism for preventing sensitive input leakage by confining untrusted IME apps to predefined security policies. However, both of them require manual labeling of which pieces of data are sensitive inputs that need to be protected first.

Android UI/Text Analysis. Several studies utilize UI and text analysis for different security proposes. Chen et al. [44] achieve massive scale malware detection with apps' UI structure analysis. A similar UI structure indicates a possible repackaging relation, which can be used for malware detection instead of widely used heavy-weight program analysis. AsDroid [45] 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 [46] 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 [28] 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 [39] 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 [45], [5], [4], [22] 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 [22] 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.

X. 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. Our evaluation shows that UIPicker achieves 94.0% precision and 96.0% recall with manual validation on 500 popular apps. Our measurement results in different datasets shows that UIP data largely distributed in market apps and urgently need protection.

REFERENCES

[1] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones," vol. 57, no. 3. ACM, 2014, pp. 99–106.

[2] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," pp. 639–652, 2011.

[3] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proc. of ACM CCS'13*, 2013, pp. 611–622.

[4] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[5] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proc. of ACM CCS'12*, 2012, pp. 229–240.

[6] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proc. of ACM CCS'13*, 2013, pp. 1043–1054.

[7] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proc. of ACM CCS'10*, 2010, pp. 328–332.

[8] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "Asm: A programmable interface for extending android security," in *23th USENIX Security Symposium (USENIX Security 14)*, 2014.

[9] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android." in *Proc. of NDSS'13*, 2013.

[10] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android." in *Proc. of NDSS'12*, 2012.

[11] "Amazon online store," https://goo.gl/jYdVPr.

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

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

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

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

[16] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: Ui state inference and novel android attacks," in *23th USENIX Security Symposium (USENIX Security 14)*, 2014.

[17] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications," in *The 20th Annual Network and Distributed System Security (NDSS)*, 2013.

[18] Y. Zhou, K. Singh, and X. Jiang, "Owner-centric protection of unstructured data on smartphones," in *Trust and Trustworthy Computing*. Springer, 2014, pp. 55–73.

[19] "Natural language toolkit," http://goo.gl/qzWulA.

[20] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of android applications in droidsafe," in *Proc. of NDSS'15*, 2015.

[21] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 993–1008.

[22] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *Proc. of NDSS'14*, 2014.

[23] S. Son, D. Kim, and V. Shmatikov, "What mobile ads know about mobile users," in *Proc. of NDSS'16*, 2016.

[24] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for all! assessing user data exposure to advertising libraries on android," in *Proc. of NDSS'16*, 2016.

[25] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee, "The price of free: Privacy leakage in personalized mobile in-app ads," in *Proc. of NDSS'16*, 2016.

[26] "Monkeyrunner," http://goo.gl/AQsIQu.

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

[28] R. Xu, H. Sadi, and R. Anderson, "Whyper: Towards automating risk assessment of mobile applications," in *22th USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 539–552.

[29] "Android-apktool," https://goo.gl/UgmPXp.

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

[31] Y. Yang and J. O. Pedersen, "A comparative study on feature selection in text categorization," in *ICML*, vol. 97, 1997, pp. 412–420.

[32] "scikit-learn," http://goo.gl/mBzGUZ.

[33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[34] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "Supor: precise and scalable sensitive user input detection for android apps," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 977–992.

[35] S. Varma and R. Simon, "Bias in error estimation when using cross-validation for model selection," vol. 7, no. 1. BioMed Central, 2006, p. 1.

[36] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural computation*, vol. 13, no. 7, pp. 1443–1471, 2001.

[37] I. Rish, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22. IBM New York, 2001, pp. 41–46.

[38] "Qunaer 7.3.8," http://goo.gl/1vB2k7.

[39] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proc. of ACM CCS'14*, 2014, pp. 1354–1365.

[40] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. of ACM CCS'12*, 2012, pp. 217–228.

[41] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proc. of NDSS'14*, 2014.

[42] M. Mitchell, A.-I. A. Wang, and P. Reiher, "Cashtags: protecting the input and display of sensitive data," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 961–976.

[43] J. Chen, H. Chen, E. Bauman, Z. Lin, B. Zang, and H. Guan, "You shouldn't collect my secrets: Thwarting sensitive keystroke leakage in mobile ime apps," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 657–690.

[44] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 659–674.

[45] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction." in *Proc. of ICSE'14*, 2014, pp. 1036–1046.

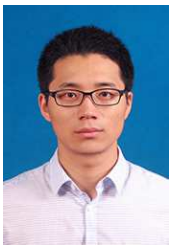
[46] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions." in *Proc. of ICSE'14*, 2014, pp. 1025–1035.

Yuhong Nan is a 3rd-year Ph.D candidate from Fudan University, Shanghai, China. He received his B.Sc in Computer Science from Hainan University in 2012. His research interests include mobile security, user privacy leakage detection/protection in mobile platforms, and program analysis techniques.

Zhemin Yang is a Lecturer with Software School, Fudan University, Shanghai, China. He received the B.Sc. and Ph.D. degrees in computer science from Fudan University, in 2007 and 2012, respectively. His research interests are in system security and program analysis techniques.

Min Yang is a Professor in Software School, Fudan University. He received the B.Sc. and Ph.D degrees in Computer Science from Fudan University in 2001 and 2006, respectively. His research interests are in system software and security.

Shunfan Zhou is a 2nd-year post-graduate student from Fudan University. He received his B.Sc in Computer Science from Fudan University in 2015. His research interests include mobile security and machine learning.



Yuan Zhang is a Lecturer in Software School, he received his Ph.D. degree from Fudan University in 2014 and B.Eng. degree from Nanjing University in 2009. His research interests include system security and compiler techniques.

Guofei Gu is an Associate Professor in the Department of Computer Science & Engineering at Texas A&M University (TAMU). He received his Ph.D. degree in Computer Science from the College of Computing, Georgia Institute of Technology. His research interests are in network and system security, social web security, cloud and software-defined networking (SDN/OpenFlow) security.

Xiaofeng Wang is a professor in the School of Informatics,Indiana University at Bloomington. He received his Ph.D. in Computer Engineering from Carnegie Mellon University in 2004. He serves as acting director of the Security Informatics Program at Indiana University since Jan. 2010. His research interest include Cloud and Mobile Security, Data and Health Informatics Security.

Limin Sun is a Professor in Institute of Information Engineering at Chinese Academy of Sciences. He received his B.S., M.S., and Ph.D degree in College of Computer, National University of Defense Technology in China in 1988, 1995, and 1998, respectively. His research interests are Internet of Things and security.