Semantics-Aware Privacy Risk Assessment Using Self-Learning Weight Assignment for Mobile Apps

Jing Chen, Chiheng Wang, Kun He, Ziming Zhao, *Member, IEEE*, Min Chen, Ruiying Du, and Gail-Joon Ahn, *Senior Member, IEEE*

Abstract—Most of the existing app vetting mechanisms only estimate risks at a coarse-grained level by analyzing apps syntax but not semantics. We propose a semantics-aware privacy risk assessment framework (SPRISK), which considers the sensitivity discrepancy of privacy-related factors at semantic level. Our framework can provide qualitative (i.e., risk level) and quantitative (i.e., risk score) assessment results, both of which help users make decisions to install an app or not. Furthermore, to find the reasonable weight distribution of each factor automatically, we exploit a self-learning weight assignment method, which is based on fuzzy clustering and knowledge dependency theory. We implement a prototype system and evaluate the effectiveness of SPRISK with 192,445 normal apps and 7,111 malicious apps. A measurement study further reveals some interesting findings, such as the privacy risk distribution of Google Play Store, the diversity of official and unofficial marketplaces, which provide insights into understanding the seriousness of privacy threat in the Android ecosystem.

Index Terms—Android, Semantics-aware, Self-learning Weight Assignment, Privacy Risk Assessment

1 Introduction

Mobile computing has gained unprecedented popularity, since users can run all sorts of applications on mobile devices to play, communicate, and work whenever and wherever they want. Almost all users store sensitive data on their mobile devices, including contact lists, SMS messages, and phone numbers. Due to the affinity between users and their mobile devices [1], privacy leakage has been considered as one of the critical challenges in mobile computing and security.

Currently, Android warns users about the required permissions when an app is installed or at runtime, trusting that users can make the right decision. However, this approach is inefficient to indicate potential privacy risks since most users do not have enough technical knowledge and patience to infer the potential privacy risks from the permission descriptions. As a result, users prefer to allow the required permissions and ignore its implications. Furthermore, privacy risks can be introduced by malicious behaviors which cannot be handled by checking permissions.

To help users understand potential privacy risks intuitively, a number of approaches have been proposed to measure apps' risks and present the measured risks to users. Most of the existing solutions only classify risks at a coarse-grained level. For example, a binary risk indicator is used to label each app as either normal or not [2]–[6]. These solutions confuse users and are not practical to help them make decisions, because apps with similar functions often have the same risk level. As a more informative indicator, a detailed score is provided to distinguish the risks of homogeneous apps. For example, Peng et al. utilize probabilistic generative models to develop risk scoring functions based on permissions requested by apps [7], but it does not consider other factors. RiskMon [8] combines users' coarse expectations and runtime behaviors of trusted apps to generate a risk score. However, they analyze apps at syntax level but not at semantic level.

There are two challenges in evaluating the privacy risk of an app. Firstly, determining whether an app has the possibility to disclose users' privacy is associated with multiple factors, including execution context, transmission destination, etc. Each factor has its own potential privacy risk, which is called *sensitivity* in this paper. For instance, the leakage of location information may be more dangerous than that of device ID. At the same time, we need to determine whether the data are sent out of the device, which is critical to the privacy risk assessment. Thus, we should consider more factors in both syntax and semantic levels. Secondly, the actual influences of each factor are different and the weight assignment for multiple factors is another challenging problem. Obviously, manually specifying or averaging the weights on different factors is inappropriate.

- *J. Chen is with the School of Cyber Science and Engineering, Wuhan University, Shenzhen Institute of Wuhan University, and Science and Technology on Communication Security Laboratory, China.*
  E-mail: chenjing@whu.edu.cn
- *C.H. Wang, K. He are with the School of Cyber Science and Engineering, Wuhan University, China.*
  E-mail: chwang@whu.edu.cn, milloglobe@gmail.com
- *Z.M. Zhao is with Rochester Institute of Technology, USA.*
  E-mail: zhao@mail.rit.edu
- *M. Chen is with the Huazhong University of Science and Technology, China.*
  E-mail: minchen2012@hust.edu.cn
- *R.Y. Du is with the School of Cyber Science and Engineering, Wuhan University, and the Collaborative Innovation Center of Geospatial Technology, China.*
  E-mail: duraying@whu.edu.cn
- *G.-J. Ahn is with Arizona State University, Tempe, AZ, USA and Samsung Research.*
  E-mail: gahn@asu.edu

Therefore, the privacy risk assessment should also include a reasonable and systematic weight assignment method.

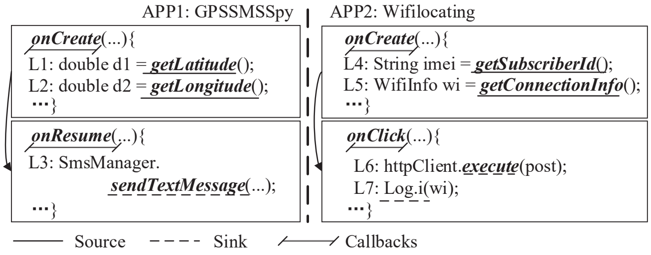
In this paper, we propose a semantics-aware privacy risk assessment framework for Android apps, called SPRisk. SPRisk considers the resource diversity in semantic level, and concerns APIs' execution context by dealing with the trigger conditions of data transmissions. Meanwhile, SPRisk also takes the transmission destination into account, while the outgoing private data has a higher risk than the one staying on the devices. SPRisk does not only offer qualitative result, *risk level*, but also provides quantitative result, *risk score*. For an app, the risk level presents a coarse-grained division, and the risk score indicates how risky the app is in a fine-grained view. Furthermore, to find the reasonable distribution of weights for various factors, we propose a novel self-learning weight assignment method, which can learn the weights from the training apps automatically.

We use SPRisk to evaluate the privacy risk distribution of real-world Android apps and malicious apps. Compared to Androguard [9], the proposed approach shows considerable improvement of accuracy. Moreover, our evaluation of SPRisk also discloses interesting and valuable findings related to risk distribution of Android marketplaces[1].

The contributions of this paper are summarized as follows:

- We propose a novel semantics-aware risk assessment framework to evaluate the privacy risk of Android apps. It deeply analyzes apps in semantic level while considering contextual API dependency correlations. Our framework provides qualitative and quantitative results, which help users understand and choose apps before installing those apps.
- Based on fuzzy clustering and knowledge dependency theory, we propose a self-learning weight assignment method to measure each factor's significance automatically. Trained with 10,000 apps, the resulting weight distribution captures the actual influences of various factors, which makes the assessment model more reasonable.
- We implement SPRisk and assess the privacy risks on 192,445 normal apps as well as 7,111 malicious apps. The experimental results demonstrate the effectiveness of our weight assignment method and the accuracy of SPRisk. A measurement study further presents some important issues, such as the risk distribution of Google Play Store and the diversity of official and unofficial marketplaces.

The remainder of the paper is organized as follows. Section 2 presents an overview of the problem and provides the architectural overview of SPRisk. Section 3 describes the design of SPRisk. Section 4 presents the self-learning weight assignment method. Section 5 presents the performance of SPRisk on a large number of samples and elaborates the experimental results, followed by a brief discussion on SPRisk in Section 6. Section 7 describes the related work. In Section 8, we conclude the paper.



```
onCreate(...){
L1: double d1 = getLatitude();
L2: double d2 = getLongitude();
...}
onResume(...){
L3: SmsManager.
        sendTextMessage(...);
...}

onCreate(...){
L4: String imei = getSubscriberId();
L5: WifiInfo wi = getConnectionInfo();
...}
onClick(...){
L6: httpClient.execute(post);
L7: Log.i(wi);
...}
```

Fig. 1: Two motivating examples.

2 Overview of SPRisk

In this section, we present an overview of the problem and the architecture of our proposed solution.

2.1 Problem Statement

In Android, the sensitive data transmission contains two necessary parts: *sources* and *sinks*, which are invoked by various callback methods. Specifically, sources mean the APIs which acquire sensitive data, and sinks indicate the APIs which deliver sensitive data. From sources to sinks, there exist some data transmission paths (including callbacks, system APIs, and other statements), called *sensitive data flows*. When assessing the privacy risk of Android apps, traditional approaches may design precise methods to find more sensitive data-flows. However, they ignore two import facts: *what the privacy is* and *how it is used*. Obviously, the leakages of different data (e.g., location and device information) have different severities, so it is more reasonable to distinguish them in a fine-grained mode. In addition, sending privacy data out of the device has a higher risk than moving them on the same device. Hence, the transmission destination may also affect the assessment result. Furthermore, it is crucial to estimate whether the data transmissions follow users' intention or not, thus, the entry points of callback methods also have an impact on the privacy risk assessment.

To motivate our work, we analyzed the sensitive data flows of two examples, as shown in Fig. 1. The first app *GPSSMSSpy* (MD5:ebae9b3a1078daa2d1a74d566780e26c) is a malware collected from Malgenomeproject [10], which accesses users' locations by sources `getLatitude()` (*L1*) and `getLongitude()` (*L2*). Then, it sends these information to remote server by a sink `sendTextMessage()` (*L3*). The data transmission is triggered in `onCreate()` and `onResume()`, which indicates that users' locations are accessed automatically. The second app *Wifilocating* (MD5:21b64328e450afad7845d1caceda26da) is an app collected from Google Play Store, which accesses device's informations (i.e., device id and WiFi configuration) by sources `getSubscriberId()` (*L4*) and `getConnectionInfo()` (*L5*). Once users click the "Search" button (`onClick()`), it sends the collected information to a remote server by `httpClient.execute()` (*L6*) and stores them locally by a sink `Log.i()` (*L7*) at the same time.

When assessing the privacy risk of these two apps, we should consider three factors: 1) the importance of data; 2) the trigger condition of data access; and 3) the distinction of data transmission. No prior work takes all three facts into

[1] The service of SPRisk is available at http://csp.whu.edu.cn/SPRisk

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/TDSC.2018.2871682, IEEE Transactions on Dependable and Secure Computing

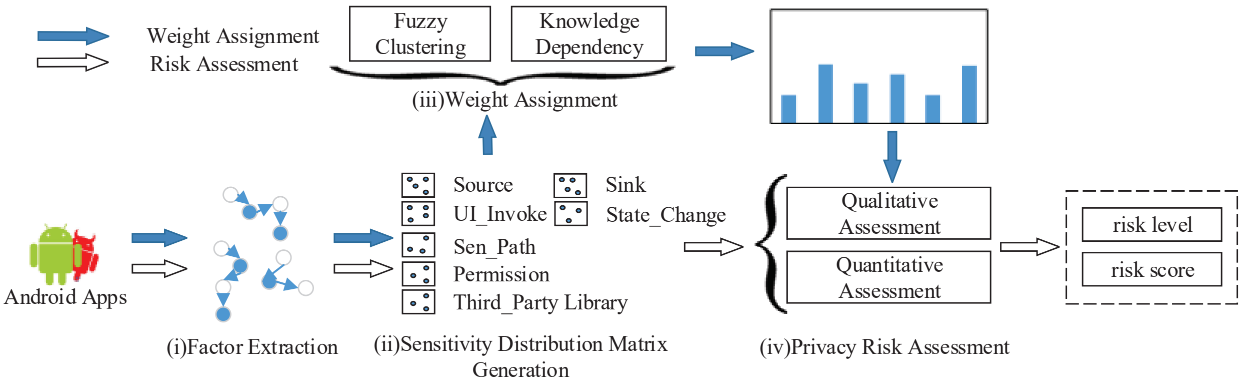


Fig. 2: Architecture overview of SPRISK.

consideration when evaluating the privacy risk of Android apps. In SPRISK, to address the aforementioned problems and complements existing vetting techniques, the following design goals are considered:

- **Semantic-based Assessment**. Our approach evaluates the privacy risk of an app in semantic level. Instead of relying on the frequency of sensitive data, SPRISK performs static program analysis for the trigger condition and destination of sensitive data flow and also considers the sensitivity of different data and permissions.
- **High Adaptability**. Our approach must be scalable to present different privacy risk levels and factors. To satisfy different demands, SPRISK needs to provide both qualitative and quantitative assessment results, which can classify privacy risk into different levels while concerning various factors.
- **Reasonable Weight Assignment**. The importance of each factor must be considered by designing a self-learning weight assignment method, which is more reasonable for evaluating apps' privacy risk. Through analyzing diverse apps in the wild, SPRISK should measure the dependency of each factor and learn the weight distribution from the training data automatically.

2.2 Architectural Overview of SPRISK

SPRISK consists of the following four major components: (i) Factor Extraction; (ii) Sensitivity Distribution Matrix Generation; (iii) Weight Assignment and (iv) Privacy Risk Assessment, as illustrated in Fig. 2.

(i) **Factor Extraction**. This component performs static program analysis to extract sensitive data transmission paths and permissions. In addition, our program analysis obtains more contextual information in semantic level. We consider the sensitivity of privacy data for providing the fine-grained assessment.

(ii) **Sensitivity Distribution Matrix Generation**. After extracting the factors, we exploit a matrix-based scheme to address the complexity challenge of multiple factors with various sensitivities. The result of this module is expressed via a sensitivity distribution matrix that describes the statistics of each factor's privacy risk.

(iii) **Weight Assignment**. Given predefined factors, this module finds the reasonable weight distribution by designing a self-learning weight assignment method. Based on fuzzy clustering and knowledge dependency theory, this module produces a weight coefficient vector, which represents the significance of each factor.

(iv) **Privacy Risk Assessment**. Once the sensitivity distribution matrix and weight coefficient vector are generated, this module evaluates the privacy risk of an app with qualitative and quantitative assessments. The risk level and risk score are provided in this module to intuitively help users understand the result.

3 DESIGN OF SPRISK

3.1 Factor Extraction

We define the *Evaluation Set* $V = \{v_1, v_2, \cdots, v_n\}$, where v_i denotes the i-th risk level. Inspired by prior works [8], [9], we define the evaluation set with a symmetric bipolar scale, namely $V = \{v_1, v_2, v_3, v_4, v_5\}$ ={`very_low`, `low`, `average`, `high`, `very_high`}, which can also help readers understand the proposed framework.

The *Factor Set* ($U = \{U_1, U_2, \cdots, U_m\}$) represents the privacy-related factors. We use a static analysis method to extract these factors from Android apps. More specifically, we focus on the sensitive data flows and the requested permissions, which can be obtained by a broad analysis over the apps' content. The detailed descriptions of these factors are as follows.

3.1.1 U_1 *Source*

In Android, a large variety of data (e.g., SMS, Contact, and Location) should be protected, and most of these sensitive data can be accessed by specific APIs. For example, in the scenario of Fig. 1, we have three source methods: `getLatitude()`, `getLongitude()`, and `getDeviceId()`. By calling each source, an app can acquire the corresponding data, e.g., calling `getLatitude()` will return the location area code. As mentioned before, different privacy data have different risks, so the sources should be assigned with different sensitivities, according to their potential privacy risks.

To have a comprehensive list of sources, we utilize SuSi [11] to extract the privacy-related data, which includes

TABLE 1: The Sensitivity Distribution of *Source*, *Sink*, and *Permission*.

Factor	Category	Sub_Category	Sensitivity
U_1 Source	PROPERTY	ACCOUNT	`very_high`
	INFORMATION	SMS_MMS CONTACT EMAIL	`high`
	DATA	LOCATION IMAGE BROWSER CALENDAR DATABASE FILE	`average`
	DEVICE	NETWORK NFC SYSTEM_SETTINGS BLUETOOTH UNIQUE_IDENTIFIER	`low`
	OTHER	SYNCHRONIZATION_DATA	`very_low`
U_2 Sink	LEAVE	NETWORK SMS_MMS BLUETOOTH VOIP SYNCHRONIZATION_DATA EMAIL NFC BROWSER PHONE_CONNECTION	`high`
	NOT_LEAVE	ACCOUNT FILE CONTACT LOCATION LOG AUDIO PHONE_STATE SETTINGS CALENDAR	`low`
U_6 Permission	CORE	ACCOUNTS CONTACT LOCATION MESSAGES CALENDAR COST_MONEY PERSONAL_INFO BOOKMARKS SOCIAL_INFO	`very_high`
	IMPORTANT	CAMERA MICROPHONE PHONE_CALLS STORAGE APP_INFO USER_DICTIONARY NETWORK VOICEMAIL	`high`
	NORMAL	AUDIO_SETTINGS BLUETOOTH_NETWORK SYNC_SETTINGS DEVICE_ALARMS	`low`

18,077 sources and 8,315 sinks. Due to the volume of this data, it is impractical to assign the sensitivity level to each API manually. In this paper, instead of assigning the sensitivity to each source individually, we consider the privacy risks of their categories. Specifically, we classify the identified sources into several categories based on SuSi, which can offer additional information about *what* has been leaked. For example, the API `getLatitude()` belongs to the category *LOCATION*, and the API `getDeviceId()` belongs to the category *IDENTIFIER*.

Next, to determine the sensitivity of each category, we manually investigate the report of DCCI [12], which ranks resources by sensitivity. For example, *ACCOUNT* is the most concerned category from the users' perspective. Thus, its sensitivity is `very_high`. Similarly, the sensitivity of category *INFORMATION* is `high`, etc. The sensitivity distribution is shown in Table 1.

Though averaging the sources' sensitivity may not be the best way towards different users' preference (i.e., users may have different views on the leakage damage of resources), the approach mentioned above is reasonable enough to demonstrate our model. Moreover, distinguishing the privacy-relevant preference of users would require a dedicated analysis for the users' cognitive experience, which is still a general, open problem for the community, and beyond the scope of this work.

3.1.2 U_2 Sink

The sinks are the potential leakage exits, which indicate the target of data transmission, such as sending SMS messages (`sendMessage()`) and writing information to log files (`Log.v()`). In this work, we classify those methods into two categories: *leave* or *not leave* the device. Hence, there are mainly two sensitivity levels for sinks. If a sink sends data out of the device, the sensitivity should be `high`. On the contrary, if data stays on the device, the data transmission has a relatively small risk and the sensitivity should be `low`.

Similarly to sources, there are also a large number of sinks in Android. Therefore, to determine the sensitivity of each sink, we also consider the API list and categories based on SuSi, as shown in Table 1. Specifically, according to the category of a sink method, we can determine *where* the corresponding data have been sent and then assign the sink's sensitivity. For example, the sink `sendMessage()` belongs to the category *SMS_MMS*, and it sends the data out of the device. Hence, its sensitivity is `high`. The sink `Log.v()` belongs to the category *FILE*. Because data stays on the device, its sensitivity is `low`.

3.1.3 U_3 UI_Invoke & U_4 State_Change

Not all sensitive data transmissions indicate a privacy leakage, it is necessary to extract whether it is users' intention to transmit. We consider the condition of a data transmission event by defining two triggers: *UI_Invoke* and *State_Change*. The *UI_Invoke* methods are related to the app's view (e.g., `onClick()`), which are typical user interactive APIs. From a security analyst's perspective, it is acceptable that private data are authorized by the user. In other words, the *UI_Invoke* method can decrease the privacy risk of an app. Conversely, the *State_Change* methods are invoked without users' approval (e.g., `onGpsStatusChanged()`). A malicious app developer commonly exploits background callbacks to perform sensitive functionalities, and increases the possibility of privacy leakage.

Unlike the sensitivity distribution of sources and sinks, we cannot determine the sensitivity of *UI_Invoke* and *State_Change* by simply using their appearance. Indeed, those trigger conditions act on the following source or sink methods. For both sources and sinks, the *UI_Invoke* can decrease the sensitivity, while the *State_Change* can increase the sensitivity. For example, assuming that the sensitivity of source is `average`, if it is called by *UI_Invoke*, the sensitivity will be decreased to `low`. However, when it is triggered by *State_Change*, the sensitivity will be increased to `high`. As a result, SPRisk can evaluate the privacy risk of an app in semantic level, rather than simply consider the sensitivity of data.

Note that, the *UI_Invoke* factor become inefficient when malware masquerades as a normal app and clicks a button before performing malicious behaviors. Therefore, simply detecting the appearance of *UI_Invoke* methods is not sufficient to declare a privacy violation. However, encoding user preference and expectations inside automated analysis is difficult, which needs the participation of users. To address this problem, we use a strict rule as follows: we consider a user-intended data transmission when an app access sensitive information with notifying the user through a prompt or license agreement. We check if a notification is displayed to the users in the causal taint tracking path from source to sink. Then, we also try to determine if the notification contains messages informing the user of data collection or requesting permission to transmit the data in question. If the condition is satisfied, the *UI_Invoke* factor is confirmed.

3.1.4 U_5 Sen_Path

We also take the whole transmission path called *Sen_Path* into account. An app usually contains multiple sensitive data flows, and there may exist many (source, sink) pairs with their own sensitivities. For example, there exist two transmission paths (`getSmsMessage()` $\rightarrow \cdots \rightarrow$ `Log.v()` and `getString()` $\rightarrow \cdots \rightarrow$ `sendMessage()`). The sensitivity of `getSmsMessage()` is higher than that of `getString()`, while `Log.v()` has a lower risk than `sendMessage()`. If we evaluate the sources and sinks individually, the inner relationship between them will be lost, and the privacy risk of the analyzed app may be inaccurate. In particular, if we map `getSmsMessage()` with `getString()` incorrectly, the app will be evaluated with a higher privacy risk level than its actual risk level. Therefore, SPRisk also treats the sensitivity transformation rule of different combinations as an indispensable factor.

There exist two transformation rules of *Sen_Path*, including remain intact and increase the risk. In fact, the sensitivity of a *Sen_Path* is based on its source, while it may be affected by the risk of its sink. For example, if the sensitivity of a sink is `low` (i.e., the privacy data has not been sent out of the device), the *Sen_Path* has the same sensitivity with its source. Otherwise, if the sensitivity of sink is `high`, the sensitivity of *Sen_Path* will be higher than that of the source. Having both factors, *UI_Invoke* and *State_Change*, the evaluation of the whole transmission path further enhances the semantics-aware capability of SPRisk.

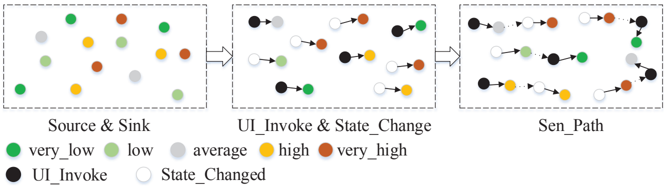


Fig. 3: The multi-level evaluation of the sensitive data transmission.

As shown in Fig. 3, our approach adopts multi-level evaluation. First, the privacy risks of sensitive data and transmission destinations are determined (U_1 and U_2), which offer information on *how many* data and channels have been accessed. Secondly, to capture the APIs' execution context, we look into the trigger conditions of each data transmission path(U_3 and U_4), which provide clues on *how* these data operations are invoked. Finally, we evaluate the sensitivity of the entire path (U_5), which is necessary to determine *what* privacy has been leaked to *where*. Although the basic measurement is the sensitivity of sources and sinks, the appearance of other factors are independent and should be treated simultaneously. In such a way, we can have a comprehensive privacy risk evaluation of the sensitive data flows in an app.

3.1.5 U_6 Permission

Permission is one of the security mechanisms in Android [13], which controls the privacy- and security-relevant parts of Android's rich API. To access sensitive data, an app should require corresponding permissions at first, and the user is notified during installation about what permissions an app will request. If developers routinely request more permissions than they require, the bug or vulnerability of the overprivileged app would increase the potential privacy risk. Thus, SPRisk treats permission as an important factor (U_6).

To implement permission assessment, we define the sensitivity by the group to which the permission belongs. In Android, each permission has been assigned to a protection level, i.e., normal, dangerous, signature, and signatureOrSystem. Unfortunately, this classification policy cannot be used for SPRisk directly. As dangerous permissions may not be related to users' privacy, there exists no congruent relationship between the protection level and the risk level. For example, the *BRICK* is a dangerous permission, which is able to disable the device. However, from the perspective of privacy protection, the possibility of leakage is small.

Therefore, we define a classification rule by the importance of permission which is determined by a large number of questionnaires [12], instead of protection levels. Specifically, we classify the permission groups into three levels: *Core*, *Important*, and *Normal*, as shown in Table 1. For example, a great percentage of users concerns the leakage of SMS, thus, the permission group *MESSAGES* belongs to the *Core* level, and the sensitivity is `very_high`. In contrast, few people worry about the leakage of Bluetooth, thus, the permission group *SENSORS* belongs to the *Normal* level, and the sensitivity is `low`.

3.1.6 U_7 Third_Party Library

Third-party libraries on Android have become a common part of apps and can provide convenience for the app developers, such as single-sign-on service, advertisement library, and online social media. However, as illustrated by a number of prior studies [6], [14], third-party libraries also increase the host apps' attack surface and have become security and privacy hazards for end users. For example, third-party libraries can leak users' SMS information, expose their host app's privileges, or track users' locations. Given the high popularity of third-party libraries and their serious impact on users' privacy, it is necessary to take the sensitivities of third-party libraries into consideration.

To detect the privacy risk of third-party libraries, the first task is to identify popular Android libraries as many as possible. In this work, we utilize LibScout [6] to build a library database that contains the ground truth of each available library version. Specifically, the collected database contains 164 libraries with 2,065 versions. In particular, advertising libraries are one of the most prevalent library types for Android, which have attracted many researchers to study the risk of them [15]–[18]. Therefore, we further utilize MAdScope [18] to retrieve 101 advertising libraries that cover the majority of existing Android advertising platforms.

Instead of relying on the number of third-party libraries, we also consider the gap between current and latest versions. As illustrated in LibScout [6], app developers only slowly adopt new library versions, exposing their end-users to large windows of vulnerability. However, not all version upgrade focuses on the repair of bugs, but in some cases tend to deliver new features. Therefore, we must define what we consider to be a bug or vulnerability upgrade. To this end, we further analyze the CHANGELOG file of each version, and look for some specific words. For example, if a new version has fixed some bugs of an old version, it may include words like “Fixed”, “Crash”, and “incorrect”. Specifically, we collect the CHANGELOG file of third-party libraries by utilizing Scrapy [19]. By checking each file been collected manually, we have saved 1,865 CHANGELOG files of 135 libraries. Through analyzing these files, we further extract static strings that typically appear in stable and test versions by parsing the CHANGELOG files. At last, we have collected 56 bug-fix related keywords and phrases in total.

When evaluating the sensitivity of third-party library, we first determine if the current version is the latest version. If not, we further determine whether the latest version has fixed some known bugs or not. To this end, we utilize LibScout [6] to identify the version of extracted third-party library. Specifically, if the version is latest one, its sensitivity should be `very_low`. If the version is not latest one and the version upgrade are only deliver new features, then its sensitivity should be `normal`. While if the version upgrade is related to bug fix, then its sensitivity should be `very_high`. For instance, the latest version of an advertising library is `v2.1.0`, while the evaluated app is equipped with `v1.2.2` version. By mapping the predefined database, we find that the latest version has fixed several bugs of old ones. Thus, the sensitivity of this current advertising library version in the evaluated app is `very_high`.

3.2 Sensitivity Distribution Matrix Generation

To represent the above-mentioned factors efficiently, we define *Sensitivity Distribution Matrix* (SDM) which can reflect the privacy risks of the data flows and permissions by a real matrix. A formal definition is presented as follows:

Definition 1. A Sensitivity Distribution Matrix ($S^{m\times n}$) is a matrix over a set of pre-defined factors and sensitivities, where:

- m denotes the number of factors, n is the number of risk levels. In this work, $m = 7$ and $n = 5$.
- In $S^{m\times n}$, each row $\mathbf{s}_i = (s_{i1}, s_{i2}, \cdots, s_{ij})$ represents the sensitivity distribution of factor i, and s_{ij} denotes the number of APIs or permissions in the context of factor U_i with the risk level v_j $(0 < j \leq n)$.

Table 2 presents an example of SDM. The sample app contains 89 sensitive data flows and requests 4 permissions. The sensitivity distribution of sources is shown in the first row. There are 89 sources in total which have different sensitivities, while 8 of them are `very_low`, 62 of them are `low`, 19 of them are `very_high`, and none of them is `average` or `high`. Hence, we can get $\mathbf{s}_1 = (8, 62, 0, 19, 0)$. By the same method, the sensitivity distribution of other factors can also be constructed.

TABLE 2: A SDM example.

Factor Set	v_1	v_2	v_3	v_4	v_5
Source	8	62	0	19	0
Sink	75	0	0	0	14
UI_Invoke	31	7	0	5	0
State_Change	0	2	5	0	0
Sen_Path	0	70	0	19	0
Permission	0	0	0	2	2
Third_Party Library	2	0	1	0	0

Among the factors mentioned above, determining the sensitivity distribution of *Sen_Path* is the non-trivial task. As introduced in Section 3.1.4, the sensitivity of each *Sen_Path* is determined by the corresponding source and sink in the data flow. Note that the sensitivities of sources and sinks are influenced by the trigger conditions. For example, there exists one data flow, `OnClick()` $\rightarrow$ `getSmsMessage()` $\cdots$ `onGpsStatusChanged()` $\rightarrow$ `log.v()`. The sensitivity of `getSmsMessage()` is `very_high`, while it is triggered by `OnClick()`, thus, the sensitivity of source is decreased to `average`. The sensitivity of `log.v()` is `average`, while it is triggered by `onGpsStatusChanged()`, thus, the sensitivity of sink is increased to `high`. As a result, the sensitivity of this data flow will be higher than that of source, i.e., increasing from `average` to `high`.

3.3 Privacy Risk Assessment

In this section, we introduce the privacy risk assessment model, which is essential to reveal the privacy risk of an app. We present a framework, which consists of two modules for qualitative and quantitative assessments. In the qualitative assessment, SPRisk attempts to examine the risk level of an app automatically, e.g., `very_high` or `very_low`. In the quantitative assessment, SPRisk takes the risk level as input, and produces the risk score of an app.

Qualitative Assessment. For the qualitative assessment, we build a single-factor evaluation matrix $\mathcal{A} = (a_{ij})^{m \times n}$ from SDM at first. $\mathcal{A}$ is constituted by the single-factor evaluation vectors. In our approach, the evaluation of each factor is denoted by a vector $a_i = (a_{i1}, a_{i2}, \cdots, a_{in})$, where $a_{ij} = \frac{s_{ij}}{\sum_{j=1}^{n} s_{ij}}$ and s_{ij} is the element of SDM.

Next, we perform the qualitative assessment by the weight coefficient vector $\mathbf{w}$ and the single-factor evaluation matrix $\mathcal{A}$ of an app. $\mathbf{w} = (w_1, w_2, \cdots, w_m)$ denotes the weight distribution of the factor set, which will be described in Section 4. In particular, the qualitative assessment result is denoted by a risk distribution vector $\mathbf{b} = (b_1, b_2, \cdots, b_n)$, where b_i indicates the possibility of each risk level v_i, and it is calculated as follows:

$$\mathbf{b} = \mathbf{w} \cdot \mathcal{A} = [w_1, w_2, \cdots, w_m] \cdot \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \tag{1}$$

The elements in vector $\mathbf{b}$ are normalized and their sum is 1. According to the principle of maximum subordination, the biggest b_i in $\mathbf{b}$ should be the result of qualitative assessment, which means that the evaluated app has the most possible with the risk level v_i.

Quantitative Assessment. For the quantitative assessment, we introduce a score vector $\mathbf{c}$, which is a n-dimensional vector $\mathbf{c} = (c_1, c_2, \cdots, c_n)$ and c_i $(0 \leq c_i \leq 100)$ is a score corresponding to each risk level v_i. The larger the score is, the higher risk an app has. In our example, $n = 5$ and we have $\mathbf{c} = (0, 20, 40, 60, 100)$. Specifically, $c_i = 0$ indicates that the analyzed app has no privacy risk at all, while $c_i = 100$ means that the possibility of privacy leakage is extremely high. Finally, we adopt the following function to calculate the risk score ϕ of an app:

$$\phi = \mathbf{b} \cdot \mathbf{c}^{\mathrm{T}} = \sum_{i=1}^{n} b_i c_i \tag{2}$$

In practice, the quantitative assessment results can help an app marketplace manager to rank the privacy risk of apps. Based on the risk ranking, the marketplace manager can design an enhanced recommendation system from the perspective of users' privacy. Moreover, SPRISK offers a significant assistance for users to decide whether a new app should be installed or not. Given several candidate apps that could provide similar services, users can choose the one which has the minimum privacy risk score.

4 Self-learning Weight Assignment

In this section, we design a self-learning weight assignment method to evaluate weights of various factors. The main idea is that when we remove a factor, the more significant the influence is, the higher weight should be assigned to the factor. To achieve this goal, our approach contains two phases: fuzzy clustering and knowledge dependency. As illustrated in Fig. 4, in the fuzzy clustering phase, SPRISK takes apps as input and classifies the dataset by the complete factor set automatically. In the knowledge dependency phase, our approach performs the classification by removing different single factor in turn. As a result, we can obtain the dependency between factors and factor sets to generate a weight coefficient vector.

4.1 Fuzzy Clustering

To classify apps into different risk levels, we first design a clustering method. The discrepancies of various privacy risks are vague, there is no clear boundary existing. Therefore, our first module, called *Fuzzy Clustering*, is based on the soft clustering method. In soft clustering, data elements can belong to more than one cluster. More specifically, the process can be divided into three stages, as in Fig. 4.

Stage 1: Data pre-processing (Step: ①). The main task in this stage is to process the original data to meet the requirements of fuzzy clustering. Let $X = \{x_1, x_2, \cdots, x_u\}$ be the dataset, and u is the number of samples. As mentioned in Section 3.2, we represent each app x_i by a SDM. Then, we quantify the risk score x_{ij} of each factor U_j for the sample x_i as

$$x_{ij} = \sum_{k=1}^{n} a_{jk} c_k \qquad (j = 1, 2, \cdots, m) \tag{3}$$

where a_{jk} is the element of the single-factor evaluation matrix $\mathcal{A}$ of sample x_i, and c_k is the element of score vector $\mathbf{c}$ introduced in Section 3.3.

As a result, X is represented by a real matrix $R = (x_{ij})^{u \times m}$. Then, we continue to normalize each element x_{ij} into $[0, 1]$ as

$$x_{ij}^{'} = \frac{x_{ij} - \bar{x}_j}{t_j} \qquad (i = 1, 2, \cdots, u; j = 1, 2, \cdots, m) \tag{4}$$

where $\bar{x}_j = \frac{1}{u} \sum_{i=1}^{u} x_{ij}$ and $t_j = \sqrt{\frac{1}{u} \sum_{i=1}^{u} (x_{ij} - \bar{x}_j)^2}$.

Stage 2: Fuzzy equivalence matrix generation (Step: ② - ③). From Fig. 4, we observe that this stage contains two steps. Firstly, we transform the matrix R to a fuzzy similarity matrix $\widetilde{R} = (r_{ij})^{u \times u}$, where r_{ij} denotes the similarity between two samples x_i and x_j. Formally, this similarity r_{ij} can be defined as

$$r_{ij} = \frac{\sum_{k=1}^{m} |x_{ik}^{'} - \bar{x}_i||x_{jk}^{'} - \bar{x}_j|}{\sqrt{\sum_{k=1}^{m} (x_{ik}^{'} - \bar{x}_i)^2} \cdot \sqrt{\sum_{k=1}^{m} (x_{jk}^{'} - \bar{x}_j)^2}} \tag{5}$$

where $\bar{x}_i = \frac{1}{m} \sum_{k=1}^{m} x_{ik}^{'}$ and $\bar{x}_j = \frac{1}{m} \sum_{k=1}^{m} x_{jk}^{'}$.

Secondly, we obtain the fuzzy equivalence matrix R^* from $\widetilde{R}$. To be specific, we find the least k $(k \in \mathbb{N})$ that satisfies $\widetilde{R}^{(2^k)} \cdot \widetilde{R}^{(2^k)} = \widetilde{R}^{(2^k)}$. Then, $R^* = \widetilde{R}^{(2^k)}$. Note that $R^* = (\delta_{ij})^{u \times u}$ is a symmetry matrix, $\delta_{ij} = 1$ $(i = j)$, and $\delta_{ij} < 1$ $(i \neq j)$.

Stage 3: Data classification (Step: ④). In this stage, we deduce the classification result based on R^*. Let λ $(0 \leq \lambda \leq 1)$ denote the membership of an app to a certain risk level, which controls the classification criteria. If $\delta_{ij} > \lambda$, the samples x_i and x_j should be classified into one level. Intuitively, through adjusting the value of λ, we can have different clustering results. In the most rigorous case $(\lambda = 1)$, every

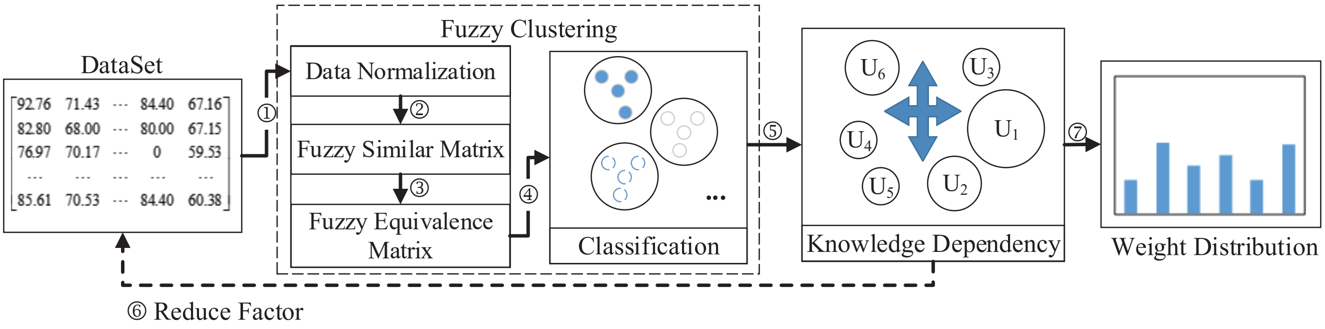


Fig. 4: The weight assignment pipeline.

app is self-contained and different apps are never classified into one risk level. When we relax the restriction (e.g., $\lambda = 0.8$), the samples, whose similarity are greater than 80% will fall into one cluster.

To find the optimal classification, we set different λ and measure the diversities of results. Our approach utilizes F-test [20] to achieve this goal, which shows the ratio between inner-group mean square deviation and intra-group mean square deviation. Let r denote the number of classes which are generated by λ. n_j indicates the number of samples in the j-th class. The samples in the j-th class is labeled as $x_1^{(j)}, x_2^{(j)}, \cdots, x_{n_j}^{(j)}$. Thus, the cluster center of the j-th class can be represented as $\bar{x}^{(j)} = (\bar{x}_1^{(j)}, \bar{x}_2^{(j)}, \cdots, \bar{x}_m^{(j)})$, where $\bar{x}_k^{(j)}$ is the average value of the k-th feature, thus

$$\bar{x}_k^{(j)} = \frac{1}{n_j} \sum_{i=1}^{n_j} x_{ik}^{(j)} \quad (k = 1, 2, \cdots, m)$$

Formally, the F-test can be defined as

$$F = \frac{\sum_{j=1}^{r} n_j \parallel \bar{x}^{(j)} - \bar{x} \parallel^2 / (r-1)}{\sum_{j=1}^{r} \sum_{i=1}^{n_j} \parallel x_i^{(j)} - \bar{x}^{(j)} \parallel^2 / (n-r)}$$

where

$$\parallel \bar{x}^{(j)} - \bar{x} \parallel = \sqrt{\sum_{k=1}^{m} (\bar{x}_k{}^{(j)} - \bar{x}_k)^2}$$

The larger the F-test is, the better the classification is. In fact, a large F-test indicates that the inner-group difference is great, while the intra-group difference is small.

4.2 Knowledge Dependency

In this section, we measure the factor dependency, namely factor weight, based on the knowledge dependency theory [21]. It provides a measurement to evaluate the dependency of different attribute subsets. The process is depicted in Algorithm 1, which takes the dataset X as input and produces the weight coefficient vector $\mathbf{w}$.

First of all, we leverage the clustering method (i.e., $equClassify()$) described in Section 4.1 to classify the original dataset. After this step, we obtain the optimal classification $Y = \{Y_1, Y_2, \cdots, Y_s\}$ on X.

Secondly, to determine the importance of each factor (Step ⑤), we define the concept of *Factor Dependency* (FD) as

Definition 2. Given two factor sets U and P, we say that U depends on P with a degree k ($0 \leq k \leq 1$), if and only if

$$k = \gamma(P, U) = \frac{|POS_P(U)|}{|X|} \tag{6}$$

where $POS_P(U)$ is a set of samples classified by U as well as P, and $|X|$ denotes the cardinality of the dataset. If $k = 1$, we say that U completely depends on P, which means that P is equivalent to U. If $0 < k < 1$, we say that U partially depends on P, and if $k = 0$ we say that U is completely independent of P.

Thirdly, we narrow the factor set U to C_i by removing the factor U_i, and construct a new dataset $\bar{X}$ from X (Step ⑥). Based on C_i and $\bar{X}$, we perform $equClassify()$ again to achieve the classification $E_i = \{E_1, E_2, \cdots, E_l\}$. To calculate the dependency between C_i and U, we realize the method $depDegree()$ by equation (6), i.e., $\gamma(C_i, U) =| POS_{C_i}(U) \mid / \mid X \mid$, where $POS_{C_i}(U) = \cup\{E_i, Y\}$. Then, we can obtain the importance of the factor U_i as

$$SGF(U_i) = 1 - \gamma(C_i, U) \tag{7}$$

where $\gamma(C_i, U)$ is the dependency of the remaining factor set C_i from U after removing the factor U_i.

Finally, we leverage the method $valueNorm()$ to normalize the weight coefficient vector $\mathbf{w} = (w_1, w_2, \cdots, w_m)$ in Step ⑦, i.e., $w_i = SGF(U_i)/\sum_{i=1}^{m} SGF(U_i)$. Note that it is quite easy to extend our current framework to complement more factors, such as dynamic code, binary files etc.

5 Evaluation and Measurement

5.1 Dataset

We conducted experiments with two datasets, including malware dataset and normal app dataset. We collected the malware dataset from three research projects (Drebin [3], DroidAnalytics [22], and Malgenomeproject [10]). We removed repeated samples and had 7,111 malware in total, as shown in Table 3. To the best of our knowledge, this is one of the largest datasets that has been used to evaluate the privacy risk on Android.

To collect normal app dataset, we crawled five representative marketplaces, including the official Android market (*Google Play Store* [23]), and four third-party Android markets (Anzhi, Anruan, Nduoa, and Gfan). We used PlayDrone [24] to obtain 172,445 apps from Google Play Store.

Algorithm 1: The Weight Assignment Algorithm.

```
Input: $X$ as the original dataset.
Output: $\mathbf{w}$ as the result of weight coefficient vector.
$Y \leftarrow equClassify(X, U)$
for $i = 1$ to $m$ do
    $C_i := U - \{U_i\}$
    $\bar{X} \leftarrow$ construct a new dataset from $X$
    $E_i \leftarrow equClassify(\bar{X}, C_i)$
    if $length(E) \neq 0$ then
        $\gamma \leftarrow depDegree(Y, E_i)$
    end
    $weight[i] := 1 - \gamma$
end
$\mathbf{w} \leftarrow valueNorm(weight)$
return $\mathbf{w}$
```

TABLE 3: Summary of Malware Dataset.

	Drebin	DroidAnalytics	Malgenomeproject
Number	5,560	2,258	1,260
Total	7,111		

For other four markets, we also built a crawler on the basis of Scrapy [19], and collected 5,000 samples from each market. Our resulting app corpus is described in Table 4. Note that we call "normal app" instead of "benign app", because the collected dataset may contain some malicious apps which were undetected yet. However, we focus our effort on the evaluation of privacy risk, not the detection of malware, thus those noisy apps would not be a problem for our approach.

We implemented the prototype of SPRisk on the basis of FlowDroid [25] with 2,437 lines of code (LOC) in Java. In particular, the weight assignment algorithm was implemented in Java and Matlab together, which introduces 924 LOCs.

5.2 Effectiveness of SPRisk

5.2.1 Weight Assignment

To examine the effectiveness of our self-learning weight assignment method, we randomly selected 5,000 apps from malware and normal app dataset, respectively. Then, we converted the selected dataset into a matrix (see Section 4). However, this way may induce unreasonable computation and storage costs (e.g., 10,000 apps need to construct a 10000 × 10000 matrix), which is unpractical. To solve this problem, we leveraged a divide-and-conquer method to save memory. In detail, the dataset was divided into 10 groups, each group has 1,000 apps. Because all the weight coefficient vectors from different groups have the same influence, we could average them to achieve the final result.

TABLE 4: Summary of Normal App Dataset.

	Official	Unofficial			
	Google Play Store	Anzhi	Anruan	Nduoa	Gfan
Number	172,445	5,000	5,000	5,000	5,000
Total	192,445				

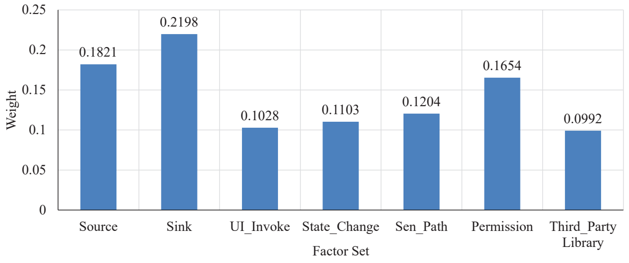


Fig. 5: Weight distribution of the factor set.

As shown in Fig. 5, the two most important factors are *Source* (U_1) and *Sink* (U_2). We compared the sensitivity distribution of these two factors by selecting 1,000 apps with `high` and `low` risk, respectively. In these `high` (`low`) risk apps, there are 2,346 (1,621) sinks and 80% (82%) of them have the sensitivity `high` (`low`). Compared with sinks, there are 1,865 (1,041) sources and only 51% (46%) of them have the sensitivity `high` (`low`). On one hand, the sensitivity distribution of *Source* is more balanced than that of *Sink*, and we cannot decide the privacy risk according to the sensitivity of *Source*. On the other hand, the sensitivity of *Sink* has a strong relationship with the privacy risk of an app. In other words, if an app has many sinks that have the sensitivity of `high`, it would have a high possibility of privacy leakage, which means that the destination of the sensitive data flow has a great influence on the privacy risk of an app.

For the privacy risk of permissions, many researchers have suggested that developers should follow the least-privilege principle and request only necessary permissions. In our approach, permission is also an impactful factor, which has the higher weight than the other four factors (U_3, U_4, U_5, and U_7). As illustrated in Section 3.1.5, different permissions have distinct influence on the privacy risk.

5.2.2 Accuracy

Note that mobile users have distinct security preference, there is no unified criterion for privacy risk assessment of apps. Therefore, evaluating accuracy is a challenging task since manually running all apps and examining each user's expected appropriate behaviors are not feasible. In this experiment, we opt for an approximation of accuracy for SPRisk. We want to evaluate how well SPRisk is able to assigning high scores to malware apps and low scores to normal apps. To this end, we randomly selected 5,000 apps from malware and normal app dataset, respectively. Fig. 6 summarizes the characteristics of the risk distribution.

We can find that most apps in the malware dataset (SPRisk-M) have higher risk than that in the normal app dataset (SPRisk-N). Especially, in the malware dataset, the percentages of apps that have risk scores higher than 60 are 73.72%. For all apps been labeled as `very_high`, 90.79% come from the malware dataset, which illustrates that our approach can obtain equivalent results. Moreover, there are 9.96% apps in the malware dataset with risk scores lower than 40. This is because that some malware samples do not aim at stealing users' private data. Therefore, the privacy

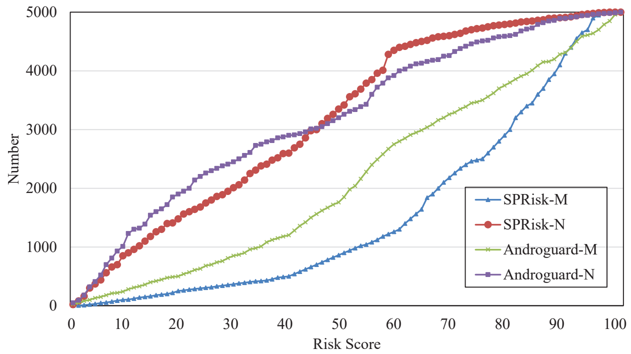
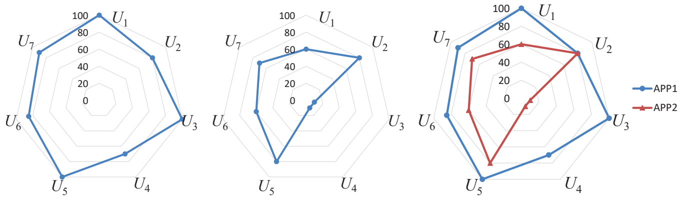


Fig. 6: The cumulative privacy risk distribution of malware and normal apps.



(a) APP1 (b) APP2 (c) APP1 VS. APP2

Fig. 7: The sensitivity distributions of two cases and their comparison.

risk scores of them are low. On the other hand, 12.65% apps have risk scores higher than 60 in the normal app dataset. In other words, privacy risk is a ubiquitous problem in most apps.

We compared the accuracy of SPRisk with Androguard [9], which is a popular static analysis tool. Androguard also provides a toolkit to evaluate the risk of an app, and the evaluation result is represented by a number between 0 and 100. Fig. 6 also shows the privacy risk distribution for evaluating malware (Androguard-M) and normal (Androguard-N) apps by Androguard. From our experiments, we find that more than 56%(2,815) of malware apps are labeled with risk scores under 60 and less than 38%(1,956) of normal apps are scored under 20. The results indicate that Androguard cannot provide appropriate evaluation for many malware and normal apps. Moreover, compared with SPRisk, we find that the gap between malware and normal apps in Androguard is little, which means that Androguard cannot differentiate the semantic of sensitive behaviors. Through analyzing the source code of Androguard, we find that its risk evaluation model is too simple. It only chooses partial factors (Permissions, DexClassLoader, and Binary Files), and it does not consider the weight distribution of each factor.

5.2.3 *Case Studies*

In this section, we constructed several cases to evaluate the rationality of SPRisk. In particular, we selected two apps mentioned in Section 2 (i.e., *GPSSMSSpy* and *Wifilocating*), and performed the privacy risk assessment. Fig. 7 illustrates the results in the form of hexagon. Each vertex of the hexagon denotes the risk score of a factor, which is calculated by the equation (3) in Section 4.1.

In the first app, locations are sent out of the device and the data are accessed automatically, which means that the process is transparent to the user. Therefore, the probability of leakage is relatively high and this stealthy manner should be allocated with high risk. Fig. 7(a) shows the evaluation result of this app. The final risk level is `very_high`, and the risk score is 94.9694. The risk assessment of the second app is more complicated than the first one, since there are multiple sensitive data flows with different sensitivities. Both `getSubscriberId()` and `getConnectionInfo()` have less severity than `getLatitude()` and `getLongitude()`. For the sensitivity of trigger condition, `onResume()` increases the privacy risk, while `onClick()` is the opposite. As shown in Fig. 7(b), the final risk level is `normal`, and the risk score is 55.5482, which is better than that of the first one. Fig. 7(c) shows the difference between them. From this figure, we can easily compare the privacy risk diversity of them. In addition, we also find that the bigger hexagon area indicates the higher privacy risk, which is quite intuitive to compare the privacy risks of different apps.

Moreover, we also conduct a user study by recruit 50 participants in our school, and randomly assigned them into two parts. In the first part, participants were shown the permission screen that traditional Market uses. In the other part, participants were shown our qualitative and quantitative result by our approach. For each part, 50 apps were selected, including 20 malicious apps that steal user's sensitive data, 20 normal apps that access user's sensitive apps for legal functions, and 10 apps do not access user's data. Participants were asked whether they could understand the privacy risk of apps easily, and whether they could select the appropriate apps with the lowest risk. In this experiment, we mainly focused on the usability of SPRisk. This is measured by counting the number of participants who make the right decisions when select appropriate apps. At last, a total of 1,568 responses were submitted. Generally speaking, more people in the second part mentioned privacy risk concerns when they noticed the evaluation result. When we asked people in both parts to divide high, normal and low privacy risk apps, people in the second part also demonstrated a higher accuracy compared to their counterparts. This finding suggests that our approach can provide more privacy risk evaluation information and help users make decision appropriately.

5.3 Measurement Results

5.3.1 *Privacy Risk Distribution*

In this analysis, we focused on the privacy risk distribution of Google Play Store by evaluating 172,445 samples, as illustrated in Fig. 8. Specifically, the ratio of `very_low` (51,805), `low` (35,868) and `average` (62,080) samples in the dataset is 86.84%, which indicates that Google Play Store indeed makes effort to mitigate the high risk apps. For example, Google Play Store operates *Bouncer* [26] to scan an app for known malicious code, which also executes the app within a simulated environment to detect hidden malicious behavior. However, the ratio of `very_high` (7,897) and `high` (14,795) apps is 13.16%, due to the centralized role of Google Play Store, those apps can still affect a tremendous number of devices. In fact, more than 9,000 such apps have already

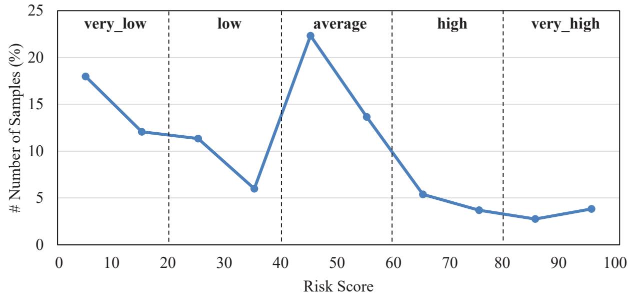
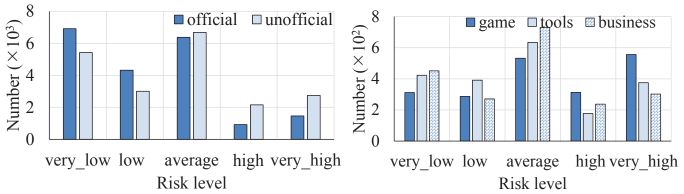


Fig. 8: The privacy risk distribution of 172,445 normal apps in Google Play Store.



(a) Official VS. Unofficial

(b) Different category

Fig. 9: Measurement results of SPRisk.

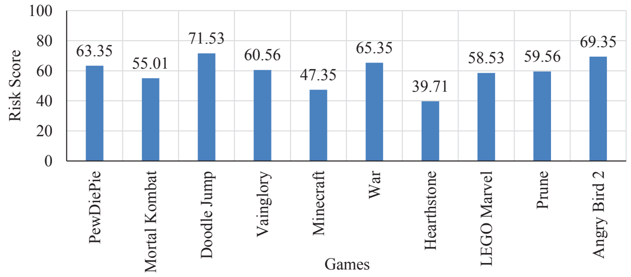


Fig. 10: The risk score rank of several Android games.

been installed over 500,000 times. Also, there are a few extremely popular ones, such as AngryBird, with the install count reaching 100 million or even more.

Although high privacy risk may not indicate that an app is malicious, we still need to pay more attention. To verify the independence of privacy risk, we scanned these `very_high` samples (7,897) in the normal app dataset by VirusTotal [27]. For each sample, we recorded the test results of different scanners. At last, only 136 apps were labeled as malware by at least two scanners.

Furthermore, we randomly selected and manually analyzed 100 apps that are not detected by VirusTotal. The analysis result shows that 21 of them are highly suspicious. On one hand, 13 apps send sensitive data out of the device (sink: `sendTextMessage()`) without any notice or warning, e.g., device information (source: `getDeviceId()`) and users' locations (source: `getLatitude`). Most seriously, these data are transmitted in plain text, which can be easily hijacked by attackers. On the other hand, 8 apps can access many high risk resources, including contacts (source: `content://contacts/data/phones`) and SMS (source: `content://sms/inbox`). Although these high risk data are not sent out of the device (sink: `log.d()`), they are accessed automatically (state_change: `onReceive()`) and the process is transparent to users. Therefore, these stealthy manners should be allocated with high risk. In our experiment, SPRisk label these apps as `very_high` risk level, which illustrates that our approach can effectively disclose the potential privacy risk of normal apps and obtain equivalent results.

5.3.2 *Official VS. Unofficial*

We further present the privacy risk diversity in *official* (i.e., Google Play Store) and *unofficial* (i.e., Anruan, Anzhi, Gfan and Nduoa) markets. We randomly selected 20,000 apps from Google Play Store and 5,000 apps from each unofficial market, respectively. The assessment results are shown in Fig. 9(a). It is evident that the average privacy risk in unofficial markets is higher than that in official market. Among those apps labeled as `very_high` (4,206), over 65.12% of them are from the unofficial markets. In addition, the number of `very_high` and `high` apps (4,893) in these unofficial markets is almost twice of that (2,391) in the official markets. This observation points out the fact that unofficial markets are lack of sufficient censorship, compared with Google Play Store. More seriously, users can download apps from any websites and copy the apk files to device, which further attracts the attention of malware authors.

Another interesting work is to rank these unofficial markets. Table 5 depicts our discovery. One can readily observe that none of the four unofficial markets can guarantee that their ratios of `very_low` and `low` apps exceed 50%. In particular, the privacy risk in Gfan is the most serious: among 5,000 apps, 842 of them are evaluated as `very_high` (16.84%) and 758 are `high` (15.16%). This can be attributed to the fact that this market exclusively focuses on releasing Android games. Many games do not need sensitive data to realize their functions, but they still attempt to collect users' data for other purposes. From this experiment, we believe that there is an urgent requirement to deploy a rigorous vetting mechanism in unofficial markets.

5.3.3 *Discrepancy among Different Categories*

As mentioned before, the app market, which mainly releases game apps, has a high privacy risk score. To further study this issue, we investigated the privacy risk discrepancy among different app categories. Without loss of generality, we focused on three common categories: 1) Games, 2) Tools, and 3) Business. For each category, we collected 2,000 samples from the normal app dataset and evaluated them by SPRisk. Fig. 9(b) shows the evaluation result. Unsurprisingly, for the category "Games", 556 (27.8%) apps' privacy risk levels are evaluated as `very_high`, and its ratio is higher than the others'. In addition, the privacy risk distribution of category "Business" is the best, and there are 1,451 (72.55%) apps with the risk level that is not higher than `average`. This is because that apps in category "Business" have few sensitive functions than that in "Games" and "Tools", which leads to a lower privacy risk. These results validate the effectiveness of our semantics-aware privacy risk assessment framework, and illustrate that SPRisk can capture the tiny discrepancy of different app types.

Finally, we evaluated the privacy risk of apps which belong to the same category and have similar utilities. For

TABLE 5: Privacy Risk Distributions of Four Unofficial Markets.

Markets	Risk level					Average score
	v_1(`very_low`)	v_2(`low`)	v_3(`average`)	v_4(`high`)	v_5(`very_high`)	
Anzhi	1,446	891	1,717	382	564	33.03
Anruan	1,603	757	1,497	528	615	33.88
Nduoa	1,211	712	1,873	486	718	35.45
Gfan	1,162	643	1,595	758	842	38.14
Total	5,422	3,003	6,682	2,154	2,739	35.13

TABLE 6: Analysis Time for Privacy Risk Assessment.

Step	Factor Extraction	Matrix Generation	Privacy Risk Assessment	Overall
Mean	138.14s	11.50s	2.47s	152.11s
Median	56.38s	7.17s	1.35s	64.9s

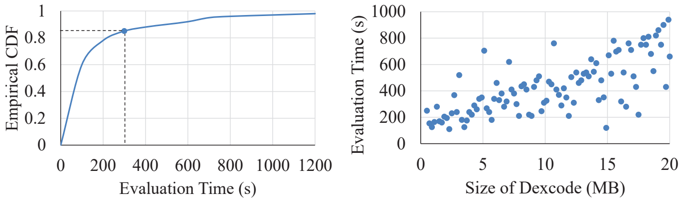
convenience, we handpicked ten Android games, including PewDiePie, Mortal Kombat, Doodle Jump, Vainglory, Minecraft, War, Hearthstone, LEGO Marvel, Prune, and Angry Bird 2. All of these games are very famous globally, and have more than 100 million players. To satisfy the individual requirement of different users, these apps always access some sensitive data. Although these behaviors may be necessary for the service, they still increase the privacy risk. We evaluated these samples through SPRisk and sorted them by their risk scores, as shown in Fig. 10. We could find that though each game has a different score, none of these games is labeled as `very_low` and all the scores are higher than 30. Especially, Doodle Jump achieves the highest risk score 71.53. Therefore, on the basis of satisfying service quality, we suggest users choose the app with the lower risk score, which can reduce the privacy risk.

5.4 Runtime Performance

To illustrate the runtime performance of SPRisk, we selected 5,000 samples from normal apps dataset. Table 6 shows the mean and median time of each step and the overall time for privacy risk assessment. SPRisk took an average of 152.11 seconds and a median of 64.9 seconds to perform the evaluation of an app. Fig. 11(a) shows the cumulative distribution of analysis time. For approximately 85% of apps, SPRisk finished the assessment within 5 minutes. As a static analysis tool, such an evaluation time is acceptable to most servers that have normal computing power.

To analyze deeply, we also evaluated the time spent with the increasing sizes of APK files. From the Fig. 11(b), we can find that the analysis time has no relation to the APK size. This occurs because the number of sensitive data transmission is independent with the scale of dexcode. In other words, a bigger APK size does not indicate more complex code logic, it may contain some big assistant files which are unrelated to privacy-risk assessment.

Moreover, we also measured the breakdown of SPRisk analysis time. In fact, the breakdown shows that around 85% to 92% of the analysis time is spent on factor extraction. We are planning to adopt a multi-threaded implementation to accelerate this phase. Meanwhile, we also discover that some data transmission paths are presented similarly and exclude these unnecessary paths could be a direction for optimization as well.



(a) CDF Distribution

(b) Time vs APKSize

Fig. 11: The runtime performance of SPRisk.

6 Discussion and Limitations

We have demonstrated that SPRisk can automatically evaluate the privacy risk of Android apps in semantic level. Obviously, except the factors mentioned above, there also exist other factors which can be extracted by static code analysis, such as dynamic code, binary files, etc. Fortunately, SPRisk is designed as a flexible solution, and it is quite easy to extend our current approach to consider more factors. To add a new factor, we only need to recalculate the weight distribution of the new *Factor Set*, and the whole evaluation model does not need any change. Accordingly, it provides a scalable approach to evaluate the privacy risk of an app, and forces future developers to design trade-offs properly to protect users' privacy.

While we have demonstrated promising results, we do not claim that our system is mature and has addressed all the problems. SPRisk chooses FlowDroid to perform static taint analysis, which turns out to be feasible and can have a good coverage of sensitive data transmission path. However, in our testing, we found that the overhead of FlowDroid is costly. Although the cost is acceptable for the server, it cannot be deployed on the device directly. To alleviate such issues, we may need to leverage more light-weight static taint analysis techniques in the future for evaluating downloaded apps directly on the device.

Furthermore, due to the code obfuscation issue of static analysis approach, there always exists the possibility that attackers find ways to evade the defender's detection by improving their technique. We expect the same to occur in our approach. To deal with this issue, we expect to combine SPRisk with dynamic analysis. With the help of automatic trigger and symbol execution techniques, the privacy risk of an intellectual app can be evaluated at runtime. Eventually, it will allow SPRisk to evaluate new variants of apps.

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/TDSC.2018.2871682, IEEE Transactions on Dependable and Secure Computing

7 Related Work

7.1 Analysis of sensitive data flows

Extracting sensitive data flows is a vivid research area in the past few years. Existing schemes can be divided into two branches. The first branch focuses on identifying more precise sensitive data flows [25], [28]–[31]. FlowDroid [25] performed context, field, object, and flow-sensitive taint analysis for Android apps. IccTA [29] sought to identify sensitive inter-component and inter-application information flows. The second branch considers more sensitive data [11], [32], [33]. For example, SuSi [11] leveraged a supervised learning approach to detect more sensitive APIs in Android platform. SUPOR [32] and UIPicker [33] automatically examined the UIs to identify sensitive user inputs that involve privacy. Different from these approaches, SPRisk well considers the risk diversities of different sensitive data flows, which can be deployed as a supplement of the existing static and dynamic analysis approaches.

7.2 Malicious app detection

There are lots of works focus on malicious app detection [1]–[4], [6], [34], [35]. To reduce manual effort, the best technique for most situations is automatic analysis. For example, TaintDroid [1] performed dynamic taint tracking of data, and tracked the sensitive data by inserting profiling code into the app. DREBIN [3] took a hybrid approach and considered both Android permissions and sensitive APIs as malware features. CHEX [36] performed static information-flow analysis to identify component hijacking vulnerabilities in Android app. DroidAPIMiner [37] extracted malware features at the API level and provided light-weight classifiers to defend against malware installations. However, the goal of these studies is to detect malware. As a result, neither of these solutions actually attempt to evaluate the privacy risk level of common apps, including benign and malicious apps. Moreover, privacy risk is an independent threat and it is not appropriate to evaluate such an app by simply utilizing existing malware detection methods.

Some researchers also took users' intention into consideration [38], [39]. For example, AUTOREB [38] explores the user review information, and utilizes the review semantics to predict the risky behaviors in Android apps. AppIntent [39] studied a method to separate user-intended Android data transmission from unintended ones. It proposes a symbolic execution approach for Android GUI applications to extract event inputs and data inputs. Inspired by these studies, SPRisk aims at helping end users to understand the privacy risk of Android apps, and therefore provides more fine-grained evaluation result.

7.3 Privacy risk assessment

There also exists some risk assessment schemes for Android apps [7], [8], [40]–[42]. For example, RiskMon [8] combined users' coarse expectations and runtime behaviors of apps to evaluate the risk of an app. It required users to provide their selection of trusted apps, thus the approach can satisfy the diverse preferences of different users. However, it analyzed apps in syntax level but not in semantic level. Peng et al. [7] proposed a permission-based risk assessment approach. They argued that a binary risk signal has significant limitations, which is consistent with our opinion. However, it only considered permissions to exploit risk scoring functions, which is superficial and needs more in-depth analysis. AppAudit [41] relied on static and dynamic analysis to provide real-time app auditing. WHYPER [42] leveraged Natural Language Processing techniques to automatically assess the risk by revealing the discrepancy between app description and the permission usage. Moreover, both of these solutions took no account of the actual influences of various factors. Our approach can address all of these problems in privacy risk assessment of Android apps.

Most related to our work is proposed by Lin et al. [40], which also assign grades to Android apps by using a privacy model they built. The privacy model measures the gap between people's expectations of an app's behavior and the app's actual behavior. Unlike them that only uses third-library as the indicator of what sensitive data that apps use and how that data is used, which could easily lose other potential channels, SPRisk utilize static taint analysis to extract sensitive data transmission flow, and ensures a good coverage of source code. Certainly, their techniques in analysis third-library could complement our factor set to further improve our approach.

8 Conclusions

In this paper, we presented SPRisk, a privacy risk assessment framework. SPRisk considers multiple factors not only in syntax level but also in semantic level. To determine the actual influence of various factors, we introduced a self-learning weight assignment method. We implemented a prototype system, which provides the qualitative and quantitative results that can intuitively help users make decisions before installing the target app. Experimental results clearly demonstrated that SPRisk is effective and feasible. With the help of SPRisk, an analyst can further reveal various helpful findings to mitigate privacy leakage in the mobile ecosystem, such as the diversity between apps with different popularities and the integrated privacy risk of a mobile device.

Acknowledgment

This research was supported in part by the National Natural Science Foundation of China under Grant Nos. 61772383, 61572380, 61702379, 61628202, the Major State Basic Research Development Program of China under Grant No. 2014CB340600. The corresponding author is Chiheng Wang.

References

[1] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 393–407.

[2] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "UNVEIL: A large-scale, automated approach to detecting ransomware," in *Proc. 25th USENIX Security Symposium*, 2016, pp. 757–772.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Efficient and explainable detection of Android malware in your pocket," in *Proc. 21th Network and Distributed System Security Symposium (NDSS)*, 2014, pp. 1–15.

[4] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual api dependency graphs," in *Proc. 21th ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 1105–1116.

[5] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. 29th Network and Distributed System Security Symposium (NDSS)*, 2012, pp. 1–15.

[6] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. 23th ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 356–367.

[7] H. Peng, C. S. Gates, B. P. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. 19th ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 241–252.

[8] Y. Jing, G. Ahn, Z. Zhao, and H. Hu, "Towards automated risk assessment and mitigation of mobile applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 5, pp. 571–584, 2015.

[9] *Androguard*, accessed on 2017. [Online]. Available: https://code.google.com/p/androguard/

[10] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012, pp. 95–109.

[11] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proc. 21th Network and Distributed System Security Symposium (NDSS)*, 2014, pp. 1–15.

[12] *DCCI Data Center of China Internet*, accessed on 2016. [Online]. Available: http://www.dcci.com.cn/media/download/5097b97e067b58b50428774fefb45c35eb89.pdf

[13] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. A. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proc. 24th USENIX Security Symposium*, 2015, pp. 499–514.

[14] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in *Proc. 23th USENIX Security Symposium*, 2014, pp. 1021–1036.

[15] 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. 23th Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 1–15.

[16] S. Son, D. Kim, and V. Shmatikov, "What mobile ads know about mobile users," in *Proc. 23th Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 1–15.

[17] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee, "The price of free: Privacy leakage in personalized mobile in-apps ads," in *Proc. 23th Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 1–15.

[18] S. Nath, "Madscope: Characterizing mobile in-app targeted ads," in *Proc. 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015, pp. 59–73.

[19] *Scrapy*, accessed on 2017. [Online]. Available: http://scrapy.org

[20] R. Lomax and D. Hahs-Vaughn, *Statistical concepts: A second course*. Vaughn, 2012.

[21] Z. Pawlak, "Rough sets," *International Journal of Computer & Information Sciences*, vol. 11, no. 5, pp. 341–356, 1982.

[22] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate Android malware," in *Proc. 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2013, pp. 163–171.

[23] *Goolge Play Store*, accessed on 2017. [Online]. Available: http://play.google.com/store?hl=en/

[24] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *Proc. ACM International Conference on Measurement and Modeling of Computer Systems*, 2014, pp. 221–233.

[25] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 259–269.

[26] *Google play: Android's bouncer can be pwned*, accessed on 2017. [Online]. Available: http://www.techrepublic.com/blog/it-security/-google-play-androids-bouncer-can-be-pwned/

[27] *VirusTotal*, accessed on 2017. [Online]. Available: https://www.virustotal.com/

[28] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proc. 22th Network and Distributed System Security Symposium (NDSS)*, 2015, pp. 1–16.

[29] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "I know what leaked in your pocket: Uncovering privacy leaks on Android apps with static taint analysis," *Eprint Arxiv*, 2014.

[30] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *Proc. 23th Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 1–15.

[31] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for android runtime," in *Proc. 23th ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 331–342.

[32] 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 *Proc. 24th USENIX Security Symposium*, 2015, pp. 977–992.

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

[34] 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. 21th ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 1354–1365.

[35] J. Chen, C. Wang, Z. Zhao, K. Chen, R. Du, and G. Ahn, "Uncovering the face of android ransomware: Characterization and real-time detection," *IEEE Transactions on Information Forensic & Security*, vol. 13, no. 5, pp. 1286–1300, 2018.

[36] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. 19th ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 229–240.

[37] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. 9th International ICST Conference on Security and Privacy in Communication Networks*, 2013, pp. 86–103.

[38] D. Kong, L. Cen, and H. Jin, "AUTOREB: Automatically understanding the review-to-behavior fidelity in Android applications," in *Proc. 22th ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 530–541.

[39] 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. 20th ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 1043–1054.

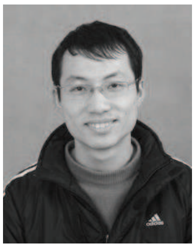
[40] J. Lin, N. M. Sadeh, S. Amini, J. Lindqvist, J. I. Hong, and J. Zhang, "Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing," in *Proc. ACM Conference on Ubiquitous Computing*, 2012, pp. 501–510.

[41] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time Android application auditing," in *Proc. 36rd IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 899–914.

[42] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proce. 22th USENIX Security Symposium*, 2013, pp. 527–542.

Jing Chen is a professor in the School of Cyber Science and Engineering, Wuhan University. He has published more than 90 research papers in many international journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Mobile Computing, IEEE Transactions on Computers, INFOCOM, SECON, TrustCom. His research interests are in the areas of network security and cloud security.

Chiheng Wang received the MS degree in computer science in 2015 from Wuhan University, Wuhan, China. He is a PH.D student of Wuhan University. His research interests include network security, mobile computing, and privacy protection.



Kun He received the PH.D degree in computer science from Wuhan University. His research interests include cryptography, network security, mobile computing, and cloud computing. He has published research papers in IEEE Transactions on Parallel and Distributed System, International Journal of Communication Systems, Security and Communication Networks, and IEEE TRUSTCOM.

Ziming Zhao received the Ph.D. degree in computer science from Arizona State University, Tempe, AZ, in 2014. He is an assistant professor in the Golisano College of Computing and Information Sciences at the Rochester Institute of Technology. He directs the CyberspACe securiTy and forensIcs lab (CactiLab). His research outcomes have appeared in the IEEE Symposium and Security and Privacy, USENIX Security, ACM CCS, NDSS, ACSAC, ACM TISSEC, IEEE TIFS, IEEE TDSC, etc.

Min Chen is a full professor in School of Computer Science and Technology at Huazhong University of Science and Technology (HUST). He was an assistant professor in School of Computer Science and Engineering at Seoul National University (SNU). He is an IEEE Senior Member since 2009. His research focuses on Cyber Physical Systems, Mobile Cloud Computing, SDN, Healthcare Big Data, Medica Cloud Privacy and Security, Body Area Networks, Emotion Communications and Robotics, etc.

Ruiying Du received the BS, MS, and PhD degrees in computer science in 1987, 1994, and 2008 from Wuhan University, Wuhan, China. She is a professor at School of Cyber Science and Engineering, Wuhan University. Her research interests include network security, wireless network, cloud computing, and mobile computing. She has published more than 80 research papers in many international journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, INFOCOM, SECON, TrustCom, and NSS.

Gail-Joon Ahn received the PhD degree in information technology from George Mason University, Fairfax, VA, in 2000. He is a professor in the School of Computing, Informatics, and Decision Systems Engineering, Ira A. Fulton Schools of Engineering and the director of Security Engineering for Future Computing Laboratory, Arizona State University. His research has been supported by the US National Science Foundation, National Security Agency, US Department of Defense, and US Department of Energy. He is a senior member of the IEEE.