International Journal of Digital Crime and Forensics
Volume 10 • Issue 3 • July-September 2018

A Privacy Protection Approach Based on Android Application's Runtime Behavior Monitor and Control

Fan Wu, Beijing University of Posts and Telecommunications, Beijing, China
Ran Sun, Beijing University of Posts and Telecommunications, Beijing, China
Wenhao Fan, Beijing University of Posts and Telecommunications, Beijing, China
Yuan'An Liu, Beijing University of Posts and Telecommunications, Beijing, China
Feng Liu, University of Chinese Academy of Sciences, Chinese Academy of Sciences, Beijing, China
Hui Lu, Guangzhou University, Guangzhou, China

ABSTRACT

This article proposes a system that focuses on Android application runtime behavior forensics. Using Linux processes, a dynamic injection and a Java function hook technology, the system is able to manipulate the runtime behavior of applications without modifying the Android framework and the application's source code. Based on this method, a privacy data protection policy that reflects users' intentions is proposed by extracting and recording the privacy data usage in applications. Moreover, an optimized random forest algorithm is proposed to reduce the policy training time. The result shows that the system realizes the functions of application runtime behavior monitor and control. An experiment on 134 widely used applications shows that the basic privacy policy could satisfy the majority of users' privacy intentions.

KEYWORDS

Android Privacy, Forensics Method, Machine Learning, Process Injection

INTRODUCTION

Background

Nowadays mobile phones are widely used in communications, and in many other aspects in people's daily lives as well. Personal information, including many kinds of privacy data, has been saved in users' mobile phones and used by installed applications on the phone. Research from Zscaler (Maritza, 2016) reveals that, in each quarter, about 0.4 percent of the mobile device transactions are leaking privacy data, including device metadata, location and personally identifiable information. In reality, many applications overuse the privilege that users grant prevalently. Privacy leaks and privilege abuse have become severe problems in mobile security. Given this, it is imperative to concern about the privacy data forensics and protection of mobile phones, since they are no longer a simple mobile device for communications, but an essential data storage tool for almost every mobile phone user.

Malware detection is a prevalent method to prevent the user's privacy leaks, which in fact contains a lot of work of digital forensics. One of the works includes detection consists of detecting

DOI: 10.4018/IJDCF.2018070108

whether applications have been repackaged, collected user privacy data or consumed fees silently. However, malware detection cannot be the only benchmark to prevent privacy leaks, since many benign applications may also abuse the users' privacy data. Furthermore, many forensic works have determined that privacy data is leaked only when its metadata is sent out of the phone, and they don't consider that privacy data might be utilized inside the applications. Mobile-phone operating systems currently provide only coarse-grained controls for regulating whether an application can access private information (Enck et al., 2014). However, they cannot control applications' behavior when a privacy leak happens.

Considering the drawbacks mentioned above, it can be concluded that privacy data is a matter of grave importance to users. Any behavior that accesses privacy data of an application is vital, and should therefore be under surveillance or provide user-notice, regardless of whether or not the application itself is malicious.

In this paper, an application behavior forensics and privacy protection system for Android based on the applications' runtime behavior is designed. The system is capable of monitoring any application's runtime behavior which accesses privacy data on Android, thus providing forensic evidence of privilege abuse and possible privacy leaks. Based on the captured runtime behavior, the system is able to enforce a rather fine-grained privacy policy that controls the source of privacy data.

RELATED WORK

Most approaches of mobile digital forensics are more or less based on applications' behavior analysis, namely the information flow tracking technique, because privacy source data is mainly utilized by various applications installed on Android devices. An application's behavior analysis mainly falls into two categories: static analysis and dynamic analysis. The former keep watch on the complete program code and all possible paths of execution before runtime, whereas the later looks at the instructions executed in the program-run in real time (Lokhande, & Dhavale, 2014).

Static Analysis

The static analysis approach requires conversion of the applications' source code before execution, for example decompiling apk files, generating Control Flow Graphs (CFG) in order to trace all possible execution paths to find any possible mal-behavior (Lokhande & Dhavale, 2014). Works related to static analysis mostly focus on malware detection and taint flow analysis. Gibler et al. (2012) proposed the AndroidLeaks system, which firstly analyzes Android framework source code to generate a map between permissions and Android APIs, then decompiles the applications' apk files; and used WALA (IBM, 2011) to track privacy information taint flow so as to find possible privacy leaks. Kim et al. (2012) proposed SCANDAL, a sound and automatic static analyzer for detecting privacy leaks in Android applications. By means of analyzing an application's Dalvik byte code, SCANDAL covered all possible states that may occur when using the application. However, it didn't make a distinction as to whether the taint flow was used for the applications' normal functions or not. All execution flows that sent the privacy data out of the system would be judged as privacy leaks. Kim et al. (2012) suggested an analysis system to prevent propagation of harmful applications. Similarly analyzing each part of an apk file statically, including dex code and Manifest file, they rated all possible risky methods, then determined the possibility of harmful applications. AppIntent (Yang et al., 2013) is another static analysis framework, which reduces the search space from static analysis, instead of searching the whole possible execution graph. Another innovation is that AppIntent determines whether the data transmission is user-intended or not, so that it may have a higher accuracy rate of judging privacy data leakage. Li et al. (2015) proposed IccTA lately, which transformed Dalvik byte code to a kind of intermediate code called Jimple, and especially focused on taint transmission between Inter-Component Communication (ICC) to find possible privacy leaks. By propagating context information among components, IccTA improved the precision of the analysis.

All static analysis methods mentioned above have some typical common ground:

- They need to analyze applications' source code.
- The searching space is rather large since it might contain all possible execution flow.
- They are able to find possible privacy data leaks in an application, but cannot prevent privacy leaking at runtime.

Batyuk et al. (2011) tried to implement privacy data confinement through a static approach. They modified an application's byte code at the source code level, controlling the applications' access of privacy data by means of changing the Android API function's return value based on a user's security preferences. The modified code was finally repackaged and installed on the user's Android phone so that some specific kind of privacy data would not be accessed by the application. However, once the application is installed, the security preference is not able to change at all.

Dynamic Analysis

Unlike static analysis, dynamic analysis has the ability to monitor the code while the code is being executed (Lokhande, & Dhavale, 2014). Crowdroid (Burguera, Zurutuza, & Nadjm-Tehrani, 2011) used a Linux application "strace" to monitor an application's system call at runtime and output its function call frequency vector, based on which it used a k-means algorithm to classify the applications. Su et al. (2012) used a similar approach to extract an application's runtime behavior, where J.48 and Random Forest classifier were used to find potential malware, which achieved high accuracy. Enck et al. (2014) proposed TaintDroid, an efficient, system-wide dynamic taint tracking and analysis system. By means of modifying the Dalvik virtual machine, it was able to mark several types of Android metadata and was capable of simultaneously tracking multiple sources of sensitive data.

In addition to malware behavior detection, there are more approaches to implement privacy data confinement by means of dynamic analysis than static analysis, since dynamic analysis is always based on an application's runtime behavior. Hornyack et al. (2011) proposed the AppFence system based on TaintDroid, which retrofitted the Android operating system. For an application's behavior of accessing privacy data, the system returned fake values so as to protect the real privacy data. However, it was not able to distinguish between the application's normal function behavior and mal-behavior. Zhou et al. (2011) designed TISSA. Modifying the Android framework, it built another permission checking system above Android's original privacy policy, which empowered users to flexibly control what kinds of personal information would be accessible to an application. Kynoid (Schreckling, Köstler, & Schaff, 2013) improved TaintDroid and was able to mark numerous kinds of metadata and defined security policies for a single data item.

As aforesaid, approaches based on TaintDroid need to modify the Android framework, real world deployment of which is not feasible. Though other approaches that use the Linux tool strace may not require system modification, they are not able to enforce data confinement during runtime but are used for mal-ware detection. Furthermore, privacy data protection policies mentioned above are absolutely based on a user's preference without being concerned about the normal functionalities of applications.

However, the authors' approach overcomes most obstacles mentioned above, as it doesn't need system modification or the applications' source code. Also, the authors propose a default privacy policy based on a large number of users' privacy intentions instead of one single user's privacy intention.

MODEL AND OVERVIEW

Overall Workflow

The authors' work mainly contains two parts: applications' runtime behavior forensics and privacy protection. The former aims at applications' runtime behavior capture and analysis, and the latter is finished based on the information produced by the former work. The system is able to capture an Android application's runtime behavior, which reveals all the sensitive behavior of an application and provides digital evidence for electronic forensics of privacy theft. Combined with users' privacy intention investigations, the runtime behavior information is used for training the privacy protection policy offline. During working, the system follows the generated privacy policy to protect the real privacy data by means of capturing and controlling applications' runtime behavior. The overall workflow is shown in Figure 1.

System Model

As shown in Figure 2, the system essentially consists of three modules: Inject Launcher, Runtime Behavior Controller and Privacy Provider. In the main user interface, the system lists all applications that are installed on the Android phone. As soon as a single application is launched through the system,

Figure 1. Overall Workflow

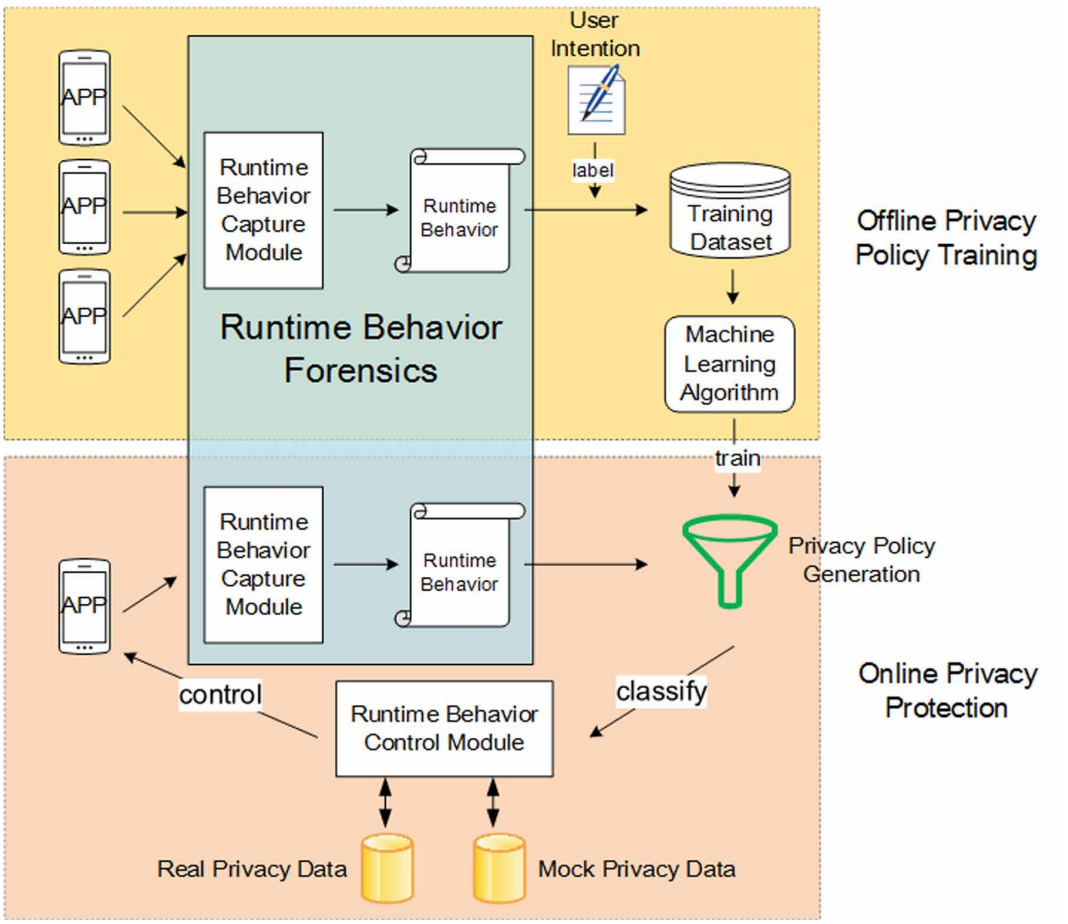
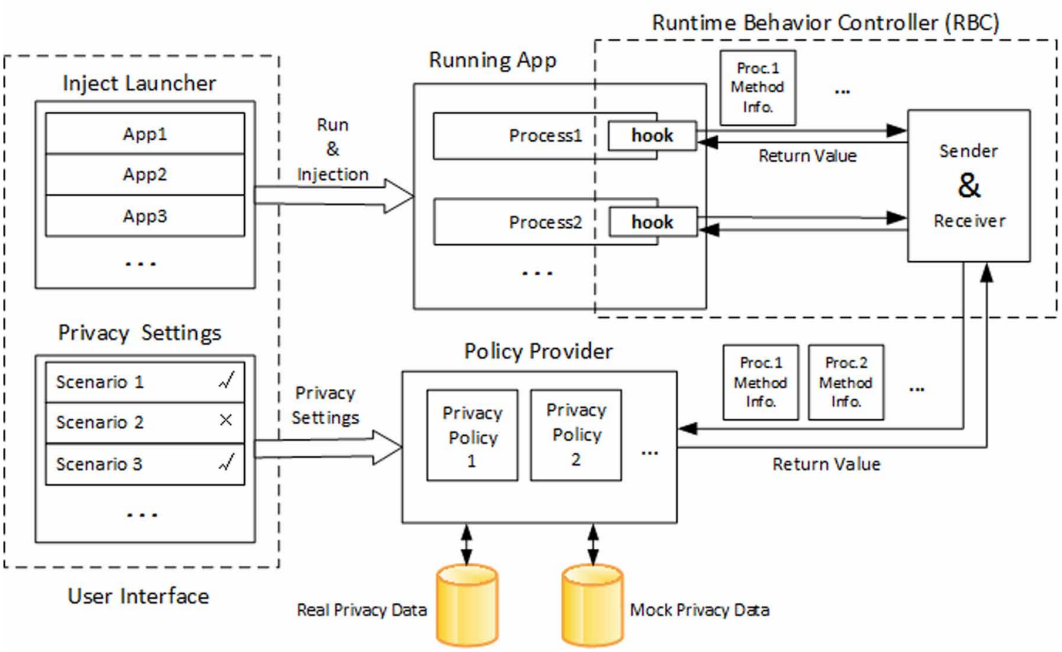


Figure 2. System Overview



the system injects a specific module into all processes that belong to the target application, redirects the sensitive functions in the application process and starts runtime behavior monitoring. When a sensitive function is called by the application, the system captures this behavior and collects some features of its calling environment, and sends the information to the Policy Provider immediately through the sender. The Policy Provider will judge whether this behavior is proper in the circumstance. According to the policy provided, the system returns the real privacy data or mock data to the application, and finally completes the policy enforcement.

Injector Launcher

The first step of the system is launching a single application installed on the Android phone and proceeding with injection in the meantime using the Linux dynamic injection technique. Dynamic injection, in essence, may allow specific dynamic link libraries (.so file in Android) to be loaded into specific address spaces of a process. Using the Linux functions *ptrace_attach* and *ptrace_detach*, the system is able to suspend and resume a certain running process. The moment the process is suspended, the system loads a dynamic link library to the memory map of the process and calls the entry function of the dynamic link library in order to start up the subsequent procedure by means of other Linux functions like *dlopen* and *dlsym*. After dynamic injection, the system has achieved attaching and starting of the Runtime Behavior Controller module to target process, which begins to monitor and control the application's runtime behavior.

Runtime Behavior Controller (RBC)

Runtime Behavior Controller (RBC) module is actually a dynamic link library complied by the Android Native Development Kit (NDK), which is injected into processes that belong to the target application after the application has been launched. It contains a Java method hook handler that controls the application's runtime behavior, and an information sender and receiver that handles the

Table 1. Sensitive APIs Related to Privacy Data

Privacy data	Behavior	API(s)
location	get	LocationManager.getLastKnownLocation(...) Location.getLatitude() Location.getLongitude() ...
phone state	get	Context.getSystemService(Context.TELEPHONY_SERVICE) TelephonyManager.getDeviceId() ...
account	get	AccountManager.getAccounts()
contacts SMS calendar call log browser history bookmarks	get	ContentResolver.query(URI)
	modify	ContentResolver.insert(...) ContentResolver.update(...) ContentResolver.delete(...) ...
user action related	get	View.performClick() Activity.onBackPressed() Activity.onPause() ...

communications between the RBC and the Privacy Provider. It has two main purposes: to capture and to control an application's runtime behavior.

Runtime Behavior Capture

Android applications are mainly implemented in Java, with the compiled class files further converted to Dalvik bytecode, running on the proprietary register-based Dalvik VM (Xu, Saïdi, & Anderson, 2012). In practice, Android applications complete a variety of functions by means of calling a series of Java Application Programming Interfaces (API). If the system is able to observe each API calling of a running application, the system is completely familiar with its runtime behavior. A Java method hook technology is employed to achieve this goal, which proceed as follows.

Privacy API Map

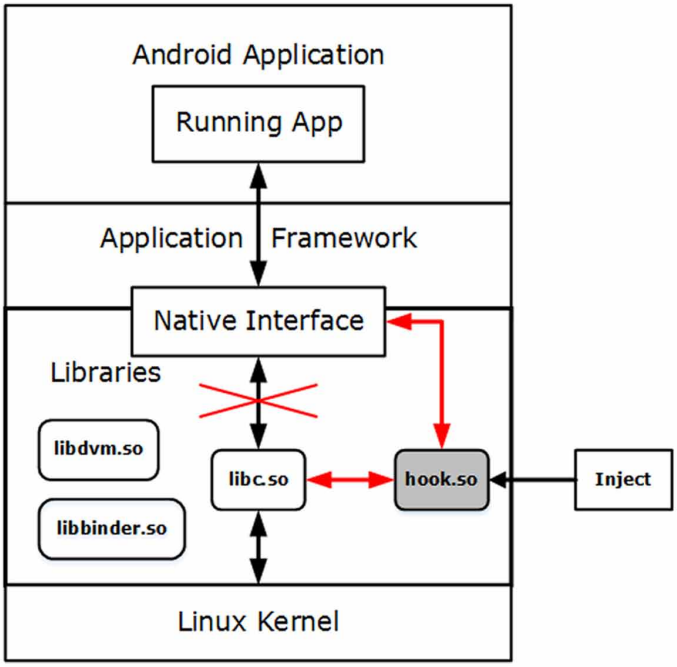
In order to capture specific behavior, the Java APIs related to the sensitive data need to be found. Pscout (Au, Zhou, Huang, & Lie, 2012) mentions many APIs associated with each kind of permission, whereas the authors here select several APIs that matter the most for accessing privacy data. With the help of the Android developers document, some kinds of application's sensitive behavior are mapped to their corresponding Java APIs in Table 1.

It can be found that functions related to user behavior are taken into account – methods which contain not only APIs of user actions such as clicking, but also some APIs of the Activity's life cycle that are probably caused by user actions. The user actions are considered to be an imperative factor for privacy policy, which might determine whether an application's behavior is user caused.

Java Method Hook

Each Java method has its fundamental method structure in the Dalvik Virtual Machine. At the Java Native Interface (JNI) level, the system locates a specific Java method's fundamental structure through its method signature using JNI APIs. By means of changing the method's native function pointer to the hook function, the so-called *method_handler*, and saving the original method pointer, the system is able to redirect the method pointer to *method_handler*. In *method_handler*, the system processes a

Figure 3. Layer of Injection and Hook



series of tasks such as runtime information collection, and then, calls the original method so as to finish the method's original functionality. Finally, the system accomplishes intercepting Java APIs without influencing their normal functionality. Figure 3 shows the layer where injection and Java method hook proceed in the Android framework structure, and illustrates the redirection of function call.

Method Runtime Information

Adjudging an application's behavior depending on no more than its calling method's name is a rather coarse-grained policy. Thus, more runtime information is taken into consideration to judge if an application's behavior is proper for privacy data extraction. A six-dimensional feature for each sensitive function call for the purpose of labeling different runtime environments is proposed:

- *method_name*: a string value, denotes the API name that is called.
- *time_from_start*: a real value, denotes the millisecond from the beginning of application runtime to the moment the method is called.
- *time_from_last_call*: a real value, denotes the millisecond from the moment of the method's last call to this call in the same thread.
- *time_from_last_click*: a real value, denotes the millisecond between the moment of the user's clicking behavior and of the moment when the method is called.
- *is_app_visible*: a boolean value, denotes whether the application is visible for the user when the method is called.
- *is_in_main_thread*: a boolean value, denotes whether the method is called by the main thread of the application.

Based on the features mentioned above, the system may distinguish between different moments when an application's behavior happens, e.g. whether the behavior happens just after the application

is launched, whether the behavior is triggered by a user's interaction with the application. Thus, privacy protection policy with a finer granularity can be implemented.

Runtime Behavior Control

The system has redirected a Java method to *method_handler*, the return value of which is supposed to be the same as the original method so as not to interfere with its normal functionality. However, the return value of a method always performs as a source of data leak, in other words, applications always access privacy data through the return values of particular APIs. Hence, it is considered to control an application's runtime behavior by means of manipulating the return value of sensitive APIs. In case of an application crashing when modifying the return value to null for all kinds of functions, the authors provide a mock value database for each category of privacy data to return a bogus value that might not have any impact on the application's normal running when the behavior is not appropriate. The real privacy data is protected by the system.

Policy Provider

The Policy Provider is aimed to manage privacy settings on the phone, the privacy database and the mock value database. Receiving method runtime information collected by the RBC, the Policy Provider determines whether or not to return the original value or bogus value to the current function call. In practice, it is a classifier capable of determining whether a function call is proper or not considering its runtime environment and the application's classification.

Basic Policy

The Basic Policy is designed for default policy settings, which means it implements privacy policy without any user's settings in order to protect privacy data automatically. It is designed to be a universal privacy protection policy that satisfies the majority of people's intentions. Thus, the dataset of Basic Policy should be big data collected from many users. A primary challenge is how to collect users' intentions and map them to a machine learning dataset. The procedure is designed as follows.

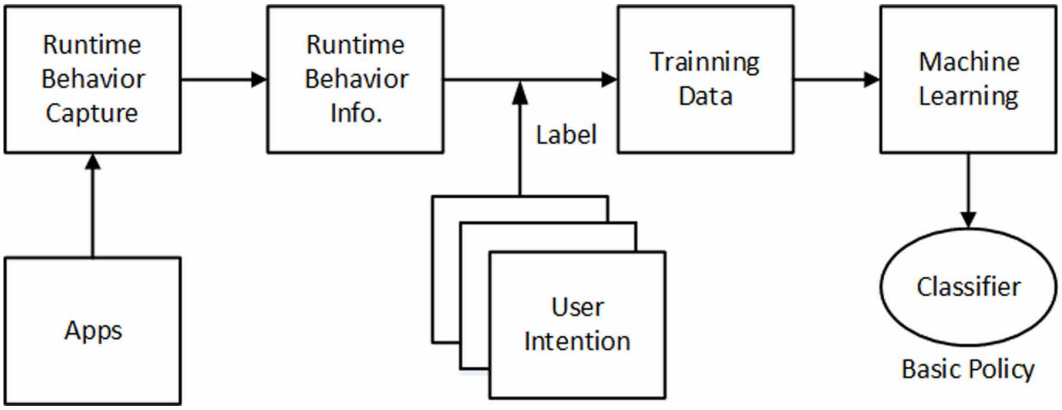
The applications' runtime behavior is classified into two levels of granularity. Firstly, for each kind of privacy data, Android applications are divided into n categories according to their privacy data needs. However, this classification is of a coarse granularity for privacy protection that may lead to massive false positives.

Secondly, m runtime scenarios are given out when privacy data is actually used during runtime. All of these m scenarios not only cover the whole life-cycle of a typical Android application's runtime, but also take the user's behavior into account, which enables a fine-grained privacy policy. Considering the boundaries between each scenario might be obscure in case of different runtime behavior of applications, so it is not feasible to determine the scenario only depending on the time value of a function call. Therefore, the six-dimensional feature for each function call using a machine learning algorithm to identify different scenarios is proposed.

In order to map user intentions to runtime scenarios, these scenarios are described as some real-world events so that users can easily understand the circumstances when their privacy data is being accessed. For example, scenarios could be described as "The application is accessing your location information while you are just using the application to read an article" or "The application extracts your location information after you turn off the screen", such that users are able to understand the real-world scenario easily. Then users may make a decision whether or not to allow the application to access privacy data in such circumstances or not. In this case, the User Intention Vector (UIV) is defined to denote a single user's intention in all scenarios in category n:

$$UIV_n = \left(a_{n1}, a_{n2}, \ldots, a_m\right)$$

Figure 4. Workflow of Generating Basic Policy



$$a_{nm} = \begin{cases} 0, & not\,allow \\ 1, & allow \end{cases}$$

where a_{nm} denotes a user's intention of scenario *m* for an application in category *n*, which equals to 1 if the user allows the application to access privacy data in scenario *m*, and 0 if the user doesn't want to share the privacy data.

Each UIV is utilized to label a set of real world applications' runtime behavior, which represents one user's decision while using the applications. A large amount of labeled runtime behavior information is collected as a training dataset for the machine learning algorithm. Finally, a classifier is obtained, namely Basic Policy, which is able to make a privacy decision for any runtime behavior. The overall workflow is illustrated in Figure 4.

Policy Training Algorithm

Several machine learning algorithms are evaluated in the later section of this paper. Since the Random Forest Algorithm produces a better performance than other tested classifiers, due to its high precision rate and high true positive rate, the Random Forest Algorithm is chosen for the Basic Policy training.

Random Forest is an ensemble of unpruned classification or regression trees created by using bootstrap samples of the training data and random feature selection in tree induction. Prediction is made by aggregating (majority voting or averaging) the predictions of the ensemble (Svetnik et al., 2003). Moreover, Random Forest does not result in over-fitting as more trees are added but produces a limited value of the generalization error (Fan et al., 2017).

Although it performs with a high precision, the time consumption of Random Forest is much greater than other machine learning algorithms. For a training set *D* of *n* instances, Random Forest generates *t* decision trees that vote for the most popular class for the final classification result. The time consumption $time_{rf}$ of training the classifier is

$$time_{rf} = T_{rf}\left(n,t\right) = t \cdot T_{tree}\left(n\right)$$

where $T_{tree}\left(n\right)$ denotes the time consumption of training one decision tree using a training set of *n* instances.

It can be noted that the time consumption of Random Forest is directly correlated with the tree number *t*, which could be of hundreds or more. Random Forest's convergence (Breiman, 2001) shows that as more trees are added, the algorithm produces a limiting value of the generalization error, the precision of which will converge. The method is considered that is able to finish up the training procedure prematurely with a slight precision loss of the final classifier. The classifier is trained as follow:

1. Randomize n training instances and divide them into *b* parts equally. Thus, each training part contains $n_b = \frac{n}{b}$ instances. Train $t_b = \frac{t}{b}$ decision trees each time.
2. Initialize training set and tree set to empty set respectively.
3. For each turn of training, add n_b instances into training set, using the same method as the Random Forest to train t_b decision trees with the training set and then adding them into tree set.
4. After each turn of training, evaluate the classifier with the whole *n* instances of training set. If the difference of precision between this turn and last turn is less than *e* (>0), stop training and set the tree set as the final classifier; otherwise save the training set and tree set, then start the next training turn.

The threshold *e* is set manually, which would affect the number of turns of training. Figure 5 summarizes the forest-building phase.

For the *i*th turn of training, the time consumption is defined as

$$T_i = T_{train}\left(n_{ti}, t_i\right) + T_{eval}\left(n, t_{ti}\right)$$

where

$$n_{ti} = \sum_{j=1}^{i} n_j = i \cdot n_b$$

denotes the number of training instances that are used in the *i*th turn,

$$t_{ti} = \sum_{j=1}^{i} t_j = i \cdot t_b$$

denotes the number of decision trees after the *i*th turn training.

$$T_{train}\left(n, t\right) = t \cdot T_{tree}\left(n\right)$$

denotes the time consumption of model building in this turn, where $T_{tree}\left(n\right)$ denotes the time consumption of training one decision tree using *n* instances.

$$T_{eval}\left(n, t\right) = n \cdot t \cdot T_0$$

Figure 5. Optimized Random Forest Algorithm

Input: training set D, training set size n, part number b, maximum decision tree number t, precision threshold e
Output: decision tree set ***TS***

```
build(D, n, b, t, e)
    randomize(D)
    initialize tree set TS
    initialize training instances set IS
    pp ← 0
    n_b ← n / b
    t_b ← ceil(t / b)
    for i ← 1 to b do
        add n_b instances into IS
        for j ← 1 to t_b do
            build one decision tree with TS and add into IS
        end for
        p ← using D to evaluate TS and get precision
        if abs(pp – p < e) then break
        else pp ← p
    end for
    return TS
```

denotes the time consumption of evaluating current model, where T_0 denotes the time consumption of classifying one instance of one decision tree.

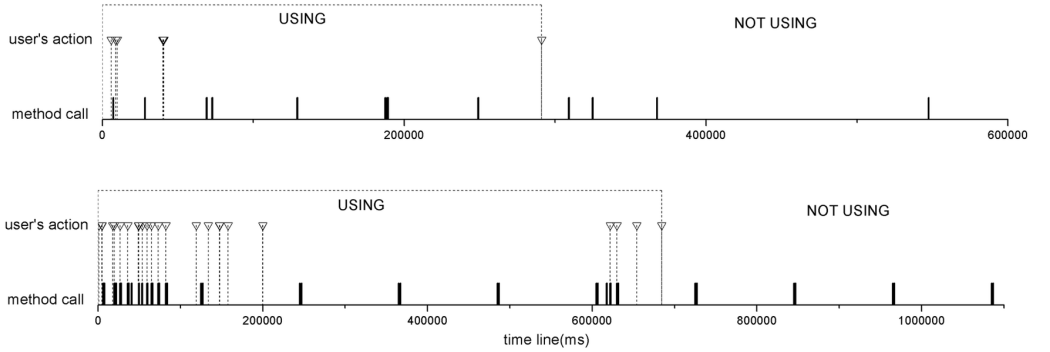
Consequently, the time consumption of the ith turn should be

$$T_i\left(n,t\right)=T_{train}\left(n_{ti},t_b\right)+T_{eval}\left(n,t_{ti}\right)=t_bT_{tree}\left(n_{ti}\right)+n\cdot t_{ti}T_0=t_bT_{tree}\left(i\cdot n_b\right)+n\cdot i\cdot t_bT_0$$

Assuming the total turns of training is r ($\leq b$), the time consumption of training the classifier $time_{new\_rf}$ is

$$\begin{aligned}time_{new_{rf}}&=T_{new_{rf}}\left(n,t\right)\\&=\sum_{i=1}^{r}T_i=\sum_{i=1}^{r}\left(t_bT_{tree}\left(i\cdot n_b\right)+n\cdot i\cdot t_bT_0\right)\\&=t_b\left(\sum_{i=1}^{r}T_{tree}\left(i\cdot n_b\right)+\sum_{i=1}^{r}i\cdot nT_0\right)\\&=t_b\left(\sum_{i=1}^{r}T_{tree}\left(i\cdot n_b\right)+\frac{r^2+r}{2}\cdot nT_0\right)\end{aligned}$$

Figure 6. Runtime Behavior of Applications from Class A



Experimental Evaluation

The authors evaluate the effectiveness of the system through its functionality on *location* privacy data. Requesting users' location information is the most prevalent behavior among Android applications, it is feasible for the authors to obtain more real-world data. Processes on other kinds of privacy data are carried out in a similar way which will not be evaluated in this paper.

Applications Dataset and Behavior Forensics

As mentioned above, the system aims at the whole privacy protection for any applications on the Android system instead of malware detection. Therefore, the applications chosen for dataset should origin from the most popular applications from the Android market. 134 benign applications in 8 categories from the Android application market that were all frequently used in people daily life were chosen to collect their runtime behavior of accessing users' location information. Furthermore, this set of applications was classified into two classes for a coarse-grained classification:

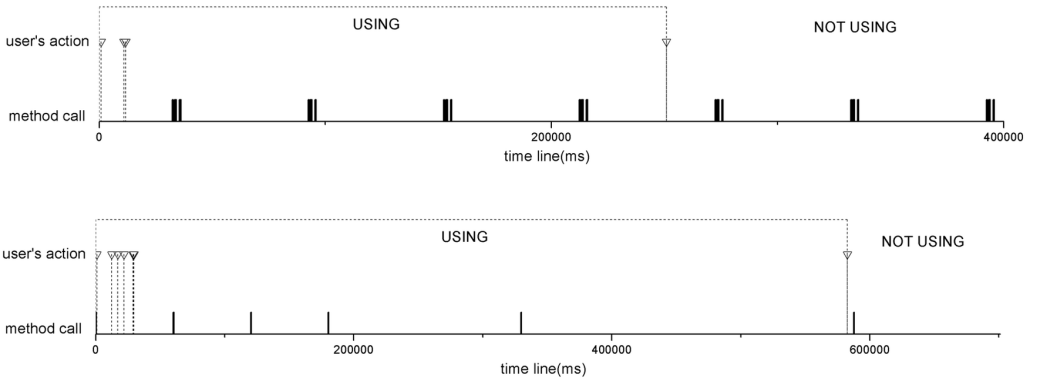
- **Class A:** Applications that might need location data on some special occasion but did not require it as a must for other normal running, e.g. applications from categories of Shopping, Weather, Travel and News.
- **Class B:** Applications that had no need of location data for their functionality, e.g. applications from categories of Tools, Music and Education.

After the classification, 74 applications were in Class A and 60 in Class B of all the 134 applications.

For behavior forensics, each application was launched from the system to capture its runtime behavior and information. The time duration of testing each application was between 5 minutes and 20 minutes including the time of user using and not using the application. In the period of time of using an application, the application was used in just the same way as everyone normally uses it. In the period of time of not using the application, the application was closed to background. Although all applications from the dataset were benign and widely used, it was found that permission overuse happened in many cases after the user gave privileges to the application.

All location accessing behavior of the applications was captured by the system shown in Figure 6 and Figure 7. The result elucidates that the application might access location information without any user action during usage time, or behave at intervals even when the application is not being used. For applications in class B, some of these also require location information even though it is not

Figure 7. Runtime Behavior of Applications from Class B



necessary for functionality. The result reflects the real behavior of the applications, which could be a great forensic evidence of privilege abuse and privacy leakage.

Basic Policy Evaluation

Overall Evaluation

The runtime behavior information of every applications from the dataset was recorded into a raw data file, which contained each sensitive function call's six-dimensional feature. The raw data file, namely pre-ARFF file, didn't contain classification information, which would be labelled by UIV and transformed to the final ARFF file. Weka (Hall et al., 2009), a powerful piece of software for data mining and machine learning, was used to generate the final classifier. The metrics used for evaluation were shown in Table 2.

The authors sent out surveys which contained several questions about four runtime scenarios to collect users' intentions:

1. **Scenario 1:** Application called the sensitive method at the beginning of the application's runtime.
2. **Scenario 2:** Application called the sensitive method just after the user interacted with the application's user interface while the user was using it.

Table 2. Definition of the Used Metrics

Term	Abbr.	Definition
True Positive	TP	Policy of allowing matches user's intention
True Negative	TN	Policy of allowing doesn't match user's intention
False Negative	FN	Policy of disallowing doesn't match user's intention
False Positive	FP	Policy of disallowing matches user's intention
True Positive Rate	TPR	TP/(TP+FN)
False Positive Rate	FPR	FP/(FP+TN)
Precision	P	(TP+TN)/(TP+TN+FP+FN)
ROC Area	AUC	Area under ROC curve

Table 3. Performance of Five Machine Leaning Algorithms

Algorithm	Class A				Class B			
	P	TPR	FPR	AUC	P	TPR	FPR	AUC
Naive Bayes	60.90%	0.609	0.296	0.661	90.21%	0.902	0.797	0.635
Bayes Net	76.04%	0.760	0.256	0.839	90.62%	0.906	0.788	0.634
SVM	61.91%	0.619	0.331	0.644	91.31%	0.913	0.913	0.500
Logistic	60.76%	0.608	0.409	0.682	91.32%	0.913	0.913	0.634
Random Forest	79.11%	0.791	0.264	0.818	91.35%	0.913	0.913	0.549

3. **Scenario 3:** Application called the sensitive method without any user action while the user was using it.
4. **Scenario 4:** Application called the sensitive method when the user was not using it but did not force to close it.

For each single survey, the authors generated one User Intention Vector (UIV), and labelled the results to a pre-ARFF file. Therefore, each data row of an application's runtime behavior had a classification, which was regarded as training set data for Basic Policy.

The dataset of Basic Policy consisted of runtime behavior of all applications mentioned before and investigations of 280 people's user privacy intention, which resulted in 238,038 metadata for the training set. Considering the difficulty of collecting users' feedback, 10-fold cross-validation was used to verify the precision of the Basic Policy. The authors partitioned the original data into 10 equal sized subsets randomly, nine of which were for training data and a single subset of which was for validation, and repeated the process 10 times with each subset used only once as the validation data. For each class of applications, the authors trained and tested five different machine learning algorithms based on 10-fold cross-validation. The overall results are shown in Table 3.

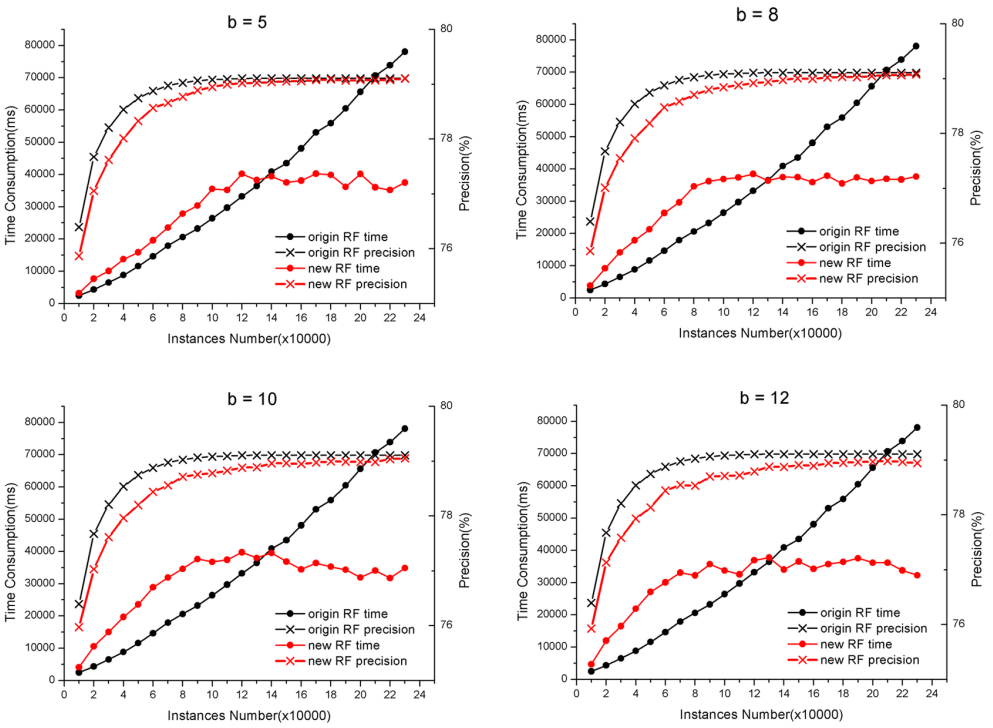
There is a big difference between the performances of the algorithms using dataset of Class A and Class B. Using the dataset of Class A, the precision ranges from 60% to 80%, while the precision is about 90% with high consistency using the dataset of Class B's applications. The root cause might be the result of user privacy intention investigation, where users' intention has more differences while judging the scenarios of applications from Class A than from Class B. Machine learning algorithms always try to output models that have high precision based on given training data, thus the classifiers can give the decision that matches most users' privacy intentions.

Optimized Algorithm Evaluation

This paper evaluated the optimized random forest algorithm on the whole dataset. At each time, 10000 randomly selected instances were added to the training set to evaluate the precision and time consumption of the classifier. The total tree number t was set to 100 and the precision threshold e was set to 0.1%.

Figure 8 illustrates the time consumption and precision of different sizes of training set in different part number b. With a small quantity training set, all parts of training dataset are trained since the difference of precision between two turns is still larger than e after all instances in the training dataset are used. However, on a larger training set, the training procedure is quit prematurely, which leads to less training time. Table 4 demonstrates that with different part number b, the optimized algorithm saves more than half the training time with little precision loss.

Figure 8. Evaluation of Optimized Algorithm



Real-World Application Evaluation

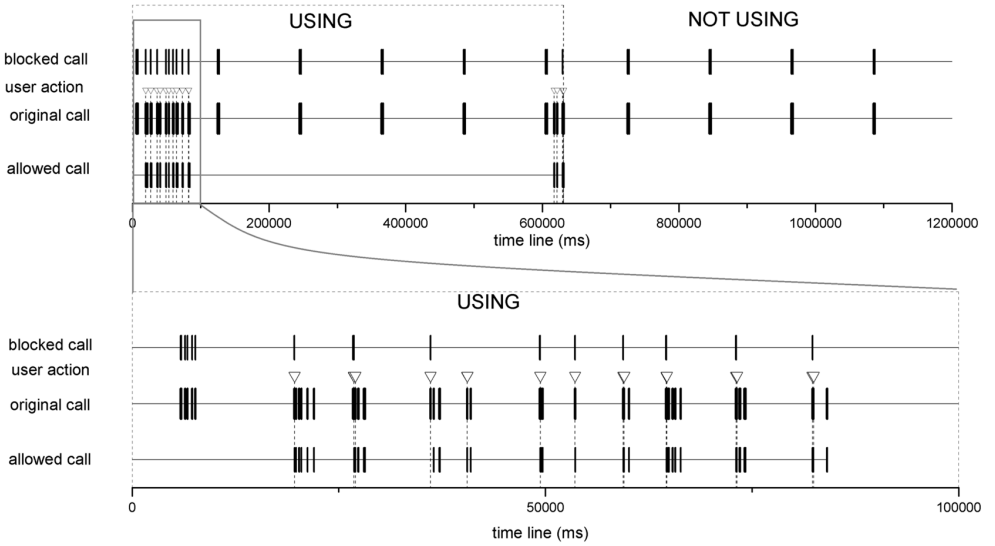
Furthermore, the Basic Policy's performance was analyzed in real world applications. Mafengwo was a travel application frequently used on Android phones that provides services such as visas, hotels, local tours and travel, and was classified in Class A. It was installed on a Nexus 4 phone with the Android 4.4 system version, running for 20 minutes and implemented with the Basic Policy. Figure 9 depicts its original behavior of requesting location information, user's click action and the system's policy implementation. Figure 9 zooms in on the beginning of usage time from 0 ms to 127100 ms, which contains details of the application's behavior while the user is interacting with the application.

Though the application declared *android.permission.ACCESS_COARSE_LOCATION* and *android.permission.ACCESS_FINE_LOCATION* in its Manifest file, it was found that the application accessed location information not only when the user was using it, but also after the user had closed

Table 4. Performance of Algorithm with Different b Values

b	Time (ms)	time/original RF time	precision (%)	precision loss (%)
5	37503.8	0.4802	79.10	0.01
8	37651.2	0.4821	79.08	0.03
10	34860.9	0.4464	79.05	0.06
12	28277.4	0.3621	78.94	0.17
original RF	78094.6	1	79.11	0

Figure 9. Basic Policy Implementation on Mafengwo



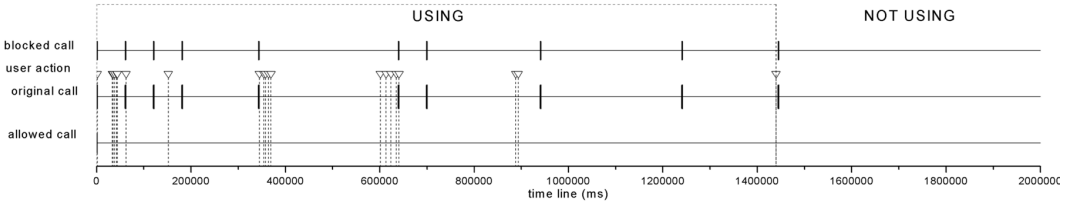
the application. The zoomed figure shows that while the user was using it, the application was still accessing location information without any user action. However, the Basic Policy blocked most of the behavior at the moment that the user was not using or interacting with the application, which indicates most users' intentions. However, a few function calls were not blocked by Basic Policy that should have been blocked apparently, such as some function calls just a few milliseconds before the user's action. The fact is that there are still omissions for the policy to identify every circumstance based on six-dimensional features of runtime behavior.

Another evaluation was carried out on a Class B application called QingtingFM, a web radio listening application with more than 3,000 radio stations nationwide. The result turned out to be quite simple, as shown in Figure 10, Basic Policy just blocked all of its location accessing behavior, which also reflected most people's privacy intention.

CONCLUSION

The authors propose a system that focuses on privacy data protection and application runtime behavior forensics on the Android system combining concepts from dynamic analysis and machine leaning. The system is able to monitor and control every sensitive behavior of an application due to different privacy policies that are based on users' intentions. The system is also used for forensic

Figure 10. Basic Policy Implementation on QingtingFM



purposes, since it is able to capture all sensitive function calls of a running application to prove its real behavior. The system is used to monitor several widely-used applications forensically, and the result shows several circumstances in which the application accesses privacy data when the user is not using it, and this can be evidence of privilege abuse. Using six-dimensional features of a single function call to distinguish different runtime environments, the system could distinguish different runtime scenarios and enforce a fine-grained privacy policy. Evaluation shows that Basic Policy could follow most people's intention to control an application's runtime behavior.

For future work, more runtime features might be proposed in order to distinguish more runtime scenarios and increase the accuracy of classification. Moreover, an incremental learning algorithm can be applied to update existing basic policy based on a user's new decisions, which can be collected by means of pop dialogs to the user when privacy data is being accessed.

REFERENCES

Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012, October). Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 217-228). ACM.

Batyuk, L., Herpich, M., Camtepe, S. A., Raddatz, K., Schmidt, A. D., & Albayrak, S. (2011, October). Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software (MALWARE)* (pp. 66-72). IEEE. doi:10.1109/MALWARE.2011.6112328

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32. doi:10.1023/A:1010933404324

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011, October). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (pp. 15-26). ACM. doi:10.1145/2046614.2046619

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B. G., Cox, L. P., & Sheth, A. N. et al. (2014). TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, *32*(2), 5. doi:10.1145/2619091

Fan, M., Liu, J., Wang, W., Li, H., Tian, Z., & Liu, T. (2017). DAPASA: Detecting android piggybacked apps through sensitive subgraph analysis. *IEEE Transactions on Information Forensics and Security*, *12*(8), 1772–1785. doi:10.1109/TIFS.2017.2687880

Gibler, C., Crussell, J., Erickson, J., & Chen, H. (2012). AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. *Trust*, *12*, 291–307.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, *11*(1), 10-18.

Hornyack, P., Han, S., Jung, J., Schechter, S., & Wetherall, D. (2011, October). These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (pp. 639-652). ACM. doi:10.1145/2046707.2046780

Kim, J., Yoon, Y., Yi, K., Shin, J., & Center, S. W. R. D. (2012). ScanDal: *Static analyzer for detecting privacy leaks in android applications.*

Kim, S., Cho, J. I., Myeong, H. W., & Lee, D. H. (2012). A study on static analysis model of mobile application for privacy protection. In *Computer Science and Convergence* (pp. 529–540). Dordrecht: Springer. doi:10.1007/978-94-007-2792-2_50

Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., & McDaniel, P. et al. (2015, May). Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering* (*Vol. 1,* pp. 280-291). IEEE Press. doi:10.1109/ICSE.2015.48

Lokhande, B., & Dhavale, S. (2014, March). Overview of information flow tracking techniques based on taint analysis for android. In *Proceedings of the 2014 International Conference on Computing for Sustainable Global Development (INDIACom)* (pp. 749-753). IEEE. doi:10.1109/IndiaCom.2014.6828062

Maritza, S. (2016). iOS, Android Apps Found Leaking User Privacy Data, Researchers Say. *Tripwire*. Retrieved from https://www.tripwire.com/state-of-security/latest-security-news/ios-android-apps-found-leaking-user-privacy-data-researchers-say/

Schreckling, D., Köstler, J., & Schaff, M. (2013). Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Information Security Technical Report*, *17*(3), 71-80.

Su, X., Chuah, M., & Tan, G. (2012, December). Smartphone dual defense protection framework: Detecting malicious applications in android markets. In *Proceedings of the 2012 Eighth International Conference on Mobile Ad-hoc and Sensor Networks (MSN)* (pp. 153-160). IEEE. doi:10.1109/MSN.2012.43

Svetnik, V., Liaw, A., Tong, C., Culberson, J. C., Sheridan, R. P., & Feuston, B. P. (2003). Random forest: A classification and regression tool for compound classification and QSAR modeling. *Journal of Chemical Information and Computer Sciences*, *43*(6), 1947–1958. doi:10.1021/ci034160g PMID:14632445

Xu, R., Saïdi, H., & Anderson, R. J. (2012, August). Aurasium: practical policy enforcement for android applications. In *USENIX Security Symposium*.

Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., & Wang, X. S. (2013, November). Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (pp. 1043-1054). ACM. doi:10.1145/2508859.2516676

Zhou, Y., Zhang, X., Jiang, X., & Freeh, V. W. (2011, June). Taming information-stealing smartphone applications (on android). In *Proceedings of the International conference on Trust and trustworthy computing* (pp. 93-107). Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-21599-5_7

Fan Wu received B.S. degree from University of Electronic Science and technology of China, Chengdu, China in 2004, and a Ph.D. degree from Beijing University of Posts and Telecommunications, Beijing, China, in 2009. Her current research interests include Privacy data protection, information security for mobile phones.

Ran Sun received his Bachelor of Engineering degree in School of Electronic Engineering from Beijing University of Posts and Telecommunications, China, in 2015. He is currently working toward the Master degree in School of Electronic Engineering from Beijing University of Posts and Telecommunications, China. His research interests include Android behavior capture and Android privacy protection.

Wenhao Fan received the B.E. and Ph.D. degree from Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2008 and 2013, respectively. He is currently an assistant professor at the School of Electronic Engineering in BUPT. His main research topics include information security for mobile smartphones, parallel computing and transmission, mobile cloud computing, and software engineering for mobile internet.

Yuan'an Liu received his M.S. and Ph.D. degree from University of Electronic Science and technology of China 1989 in 1992, respectively. He did post-doctoral research at Carleton University, Canada, from1995-1997. He has published over 500 articles in various journals and magazines. His research interests are parallel computing, wireless sensor network technology and streaming media communication.

Lu Hui received Ph.D. degree from Beijing University of Posts and Telecommunications, Beijing, China, in 2010. His current research interests include IoT security, information security for mobile phones.