SpyAware: Investigating the Privacy Leakage Signatures in App Execution Traces

Hui Xu*†, Yangfan Zhou†‡, Cuiyun Gao*, Yu Kang*†, Michael R. Lyu*
* Dept. of Computer Science, The Chinese University of Hong Kong
† MoE Key Laboratory of High Confidence Software Technologies (CUHK Sub-Lab)
‡ Dept. of Computer Science, Fudan University

***Abstract*—A new security problem on smartphones is the wide spread of spyware nested in apps, which occasionally and silently collects user's private data in the background. The state-of-the-art work for privacy leakage detection is dynamic taint analysis, which, however, suffers usability issues because it requires flashing a customized system image to track the taint propagation and consequently incurs great overhead. Through a real-world privacy leakage case study, we observe that the spyware behaviors share some common features during execution, which may further indicate a correlation between the data flow of privacy leakage and some specific features of program execution traces. In this work, we examine such a hypothesis using the newly proposed SpyAware framework, together with a customized TaintDroid as the ground truth. SpyAware includes a profiler to automatically profile app executions in binder calls and system calls, a feature extractor to extract feature vectors from execution traces, and a classifier to train and predict spyware executions based on the feature vectors. We conduct an evaluation experiment with 100 popular apps downloaded from Google Play. Experimental results show that our approach can achieve promising performance with 67.4% accuracy in detecting device id spyware executions and 78.4% in recognizing location spyware executions.**

I. Introduction

Smartphone systems are generally designed to embrace third-party applications. While such applications enrich the features of smartphones, they also bring some security and privacy issues [34]. An emerging challenge is the wide spread of spyware nested in apps, which occasionally collects data (*e.g.*, contact list, and location), and transmits them to remote servers without user's awareness [28]. These behaviors usually have a strong economic drive [16]. For example, a shopping application, or an advertiser could infer a user's interests from her browser history. The coming big data era even stimulates more needs to hunt data. Such a security issue caused by spyware on smartphone is also known as privacy leakage. It is very difficult for experts to judge whether an app leaking user's data should be deemed malicious. It depends on the features of the app, and more importantly, the user's personal preference or acceptance. Hence, traditional malware disposing approaches are not applicable for this issue, because they may either introduce many false positives or many false negatives. An intuitive idea for combating such privacy leakage issues is to reach out to user's awareness, leaving her to decide whether the leakage is acceptable.

Currently, the state-of-the-art approach for detecting privacy leakage is dynamic taint analysis, which tracks the data flow during app execution [10]. It first labels privacy-sensitive data as taint sources. Any program value whose computation depends on a taint source is also considered as tainted. In this way, the privacy leakage can be detected via monitoring whether the data being transmitted is tainted. However, such an approach generally incurs great overhead when tracking the taint propagation process. Moreover, it requires the user to flash a customized system image to replace the original operating system (*e.g.,* Android), which is risky and not practical for ordinary users.

Through a case study of privacy leakage issues with several apps, we observe that the occurrences of spyware behaviors share some common features, for example, leakages usually happen during app launch time, or when starting a new activity, and a leakage usually involves several network operations. We further infer that the data flow of leakages may have a correlation with some specific features of the program execution traces. If such a correlation can be verified, the execution trace may serve as a viable means for privacy leakage detection, which is much easier to be obtained than the taint propagation information.

To this end, we design the SpyAware framework, and adopt TaintDroid [10] as the leakage ground truth. SpyAware is mainly composed of a profiler, a feature extractor and a classifier. The profiler can instrument apps during their launch time, and capture their runtime program execution traces, including the binder calls and system calls. The traces are then separated into segments according to each user interface (UI) event. Each trace segment is defined as a `profile` of the program execution with respect to the UI event. From binder calls, the private data access behaviors can be easily identified by matching certain special calls (*e.g.,* reading the short messages usually involves calling `com.android.IContentProvider` with the parameter `content://sms`). If a profile involves a `read` behavior to some private data, the profile can be deemed as suspicious. For further evaluating, we first extract the feature vector of the suspicious profile, *i.e.,* the signature, and then let the classifier discriminate whether it indicates a spyware behavior. The features in our approach retain no app speciality, and thus the signatures from other apps are also helpful in detecting the leakage behaviors of an unstudied app. With such a cross-app detection ability, our approach does not require learning the signatures of all apps available on Google Play before hand, and thus would not suffer scalability issues.

We have conducted an experiment to evaluate the effectiveness of our approach based on 100 popular apps downloaded from Google Play. The experimental results show

978-1-5090-0406-5/15/$31.00 ©2015 IEEE

that our approach can achieve promising performance with 67.4% accuracy in detecting device id spyware behaviors and 78.4% in detecting location spyware behaviors. To our best knowledge, this paper serves as the first attempt to investigate the correlation between the data flow of privacy leakage issues with execution traces. Our tools, together with the experimental data, are publicly available to facilitate follow-up research[1].

The rest of this paper is organized as follows: Section II overviews the technical background. Section III introduces our motivation with a privacy leakage case study and the problem definition. Section IV illustrates our methodology with detailed instrumentation, feature extraction and classification methods. In Section V, we evaluate the performance of our approach. Related work is discussed in Section VI. Finally, Section VII concludes this paper.

II. Background of the Research

A. Threat Model

In this paper, we assume the following adversary model: Android apps, downloaded from Google Market and installed on smartphones, may read private data stored on the phone, and transmit such data via network. The private data can be classified into four categories listed bellow:

a) *Basic Phone Data*: The data such as call history, contact list, and SMS are related to the basic features of a classic feature phone. Android defines standard URIs (Uniform Resource Identifiers) for such data, which can be employed by applications to retrieve them (*e.g.,* com.android.phone).

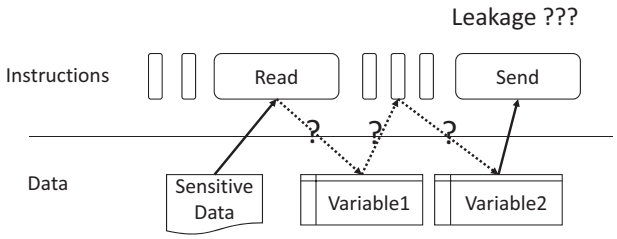
b) *Application Data*: Android allows the installations of third-party apps to enrich their features. Such applications usually have their own data stored in the phone (*e.g.,* bookmarks), which may be privacy-sensitive as well. Android also provides URI-based approach to retrieve such data. For example, a URI like 'content://browser/bookmarks' is for bookmarks.

c) *Sensory Data*: A key innovative feature of smartphones is employing various kinds of sensors to enrich the functionality of applications. General sensors include GPS, accelerometer, proximity sensor, microphone and camera. Android provides standard APIs to acquire data from them.

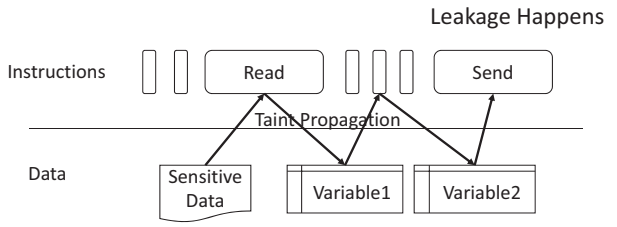
d) *Hardware Info*: This category includes information regarding the identity of the smartphone, such as IMEI (International Mobile Equipment Identity), ICCID (Integrate Circuit Card Identity) and SN (serial number). Android also provides standard APIs to retrieve them.

The leakage of the aforementioned private data would be harmful if the data are misused. For example, the SMS or some private photos can be misused by attackers to commit frauds. Note that some data we listed above seem irrelevant to a user's privacy, like IMEI; but they are. For example, IMEI can be used by service providers to uniquely track a device. To some users who are less concerned with privacy, the tracking may be acceptable. However, it may not be so to other users whose privacy is sensitive. A service provider may infer that two users have close relationship if they use the same device to login their accounts. Therefore, user's awareness about such adversaries is very important for combating privacy leakage.

[1]Project URL: http://xuhui.me/spyaware.



(a) Anti-virus software cannot detect privacy leakage issues if the data have been transformed (*e.g.,* encrypted)



(b) Dynamic taint analysis is able to track the sensitive data even if they have been transformed, and is hence effective for the privacy leak detection

Fig. 1: The limitation of anti-virus software in detecting privacy leakage

B. Limitations of Current Solutions

1) Anti-virus Software: Android adopts an installation-time permission granting mechanism, *i.e.,* users are required to accept a permission granting list declared by an app so as to install it [31]. A recent study shows only 17% Android users pay attention to the permission declaration during installation [12]. As a result, many apps declare and use permissions that are not consistent. Targeting on this security issue, some widely adopted Android security packages (*e.g.,* LBE [1] and Qihoo360 [2]) provide features to enhance the permission mechanism. They can promote users' control over the permission usage after installation by hooking into Android permission-check methods. Once a permission check is invoked, they can display a dialogue requiring user's granting or simply send a notification. Such mechanisms alleviate the permission over-privilege problem. However, they cannot know whether a permission usage will eventually cause a leakage, because they do not track the data flow of the private data. Fig. 1 illustrates such a limitation in comparison to the dynamic taint analysis approach.

2) Dynamic Taint Analysis: To analyze software behaviors, dynamic taint analysis is one of the most commonly adopted approaches. It can detect privacy leakage by tracking the information flow between sources and sinks during software runtime. Such an approach can be implemented at the instruction level, which incurs great overhead, or at higher levels which introduces some degree of capability sacrifice [10].

To our best knowledge, neither official, nor third party Android smartphone manufacturers enable the dynamic analysis feature within the source code. In order to use such a

feature, users have to download Android source code first, and then get some extra patches (*e.g.,* TaintDroid) provided by the developers. They should compile the OS source code together with the patches to build a new Android OS image with taint features enabled, and then flash the new image into their smartphones. The process is very complex, and is usually impractical for ordinary users. Moreover, it involves great risks that the smartphones would not function after flashing a new image, thus discouraging a wide adoption.

C. Android-specific Characteristics

Android is by nature a framework built on top of Linux. Apps are installed and managed through the framework. In order to simplify app development, Android provides standard system services (*e.g.,* com.android.phone) for particular features, which run as background processes. To facilitate inter-process communications (IPCs) and provide centralized security controls, Android adopts a binder-based IPC mechanism. Binder-based IPC is extensively used during application lifecycle, for example, when refreshing screen display, or when acquiring network status. Fig. 2 overviews the architecture of such a mechanism.

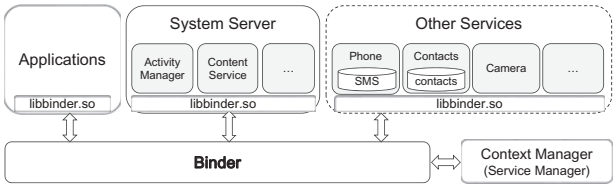


Fig. 2: Architecture of binder-based IPC

In Android, a binder is a virtual device specially tailored for IPC. Processes communicate with each other through binder calls via such a device. To connect with the binder, a process is usually required to be registered in the service manager with a unique name identifier (*i.e.,* usually a domain name) and a handler. When a process wishes to communicate with another process, it has to wrap the handler together with binder calls into a `BpBinder` object. Usually before that, the process has to look up the handler of the target process by inquiring the service manager. Hence, if we can capture such binder calls, we would be able to interpret them and get some useful information, *e.g.,* which service an app is communicating with.

III. Motivation of the Problem

A. A Privacy Leakage Case Study

To study the behavior characteristics of privacy leakage, we manually run 100 popular apps on Galaxy Nexus installed with TaintDroid. Those apps are downloaded from Google Play, and belong to several categories, including social, housing, shopping, game, *etc*. Our findings are summarized as follows:

Leakage is very common among apps: Among all those apps, 69 apps leak data. Device id (*e.g.,* ICCID, IMEI) and location are the most popular data types of leakages. Besides, we also find ten apps leaking the contact list, three leaking SMS, and one leaking bookmarks.

Reading sensitive data does not imply a leakage: In general, a leakage usually starts with a `read` behavior on private data. However, as we have discussed, it is also possible that the data would be used locally, thus not committing a leakage. During our case study, besides taint leakage notifications, we also monitor the binder calls to find such `read` behaviors. As a result, we find that such leakage-free `read` behaviors are very common in real world apps.

Some leakages are not intended by users: For several leakage cases, when the leakage notifications are reported by TaintDroid, we do not expect our operation would trigger these leakages, which means these leakages are not intended by users. For example, `com.chinamobile.contacts.im` provides features for users to backup their contacts to the server side. Users can press the backup button or activate the auto synchronization function to use this feature. However, during the experimental process, we notice that there are leakages detected by TaintDroid even when users do not press the button or activate the auto synchronization function. As a result, users cannot realize that their private data have been leaked.

A portion of the leakages happen during app launch time: Over 35 apps leak data when we launch them, which also implies that those leakages are not intended by users.

Privacy leakage usually happens when starting a new activity: We observe some leakages happen when starting a new activity. This is reasonable as a new activity usually involves some new features, which may require private data. For example, when clicking search friends button on a social app, a new activity is created, which requires the location data to recommend friends nearby.

Several leakages follow one read behavior: During our experimental process, we observe that several leakage notifications may be triggered after one single `read` behavior. Logically, it is not reasonable for an app developer to retrieve the same data (*e.g.,* IMEI) twice. We think some of these leakage notifications may be false positives because taint analysis tends to incur such problems during taint propagation process. The variables and memory spaces are likely to be contaminated. To avoid being interfered by them, in our later methodology and evaluation process, we only focus on the UI events that contain `read` behaviors.

B. Problem Definition

According to the observations in the previous case study, we find that there might be a correlation between the data flow of privacy leakage issues with some features of program executions. In other words, if certain features of the program execution are detected, it might imply a leakage has happened. Fig. 3 visualizes such a problem with four sample traces. Each of the traces contains a read behavior, and is thus deemed suspicious. The problem can be defined as whether there are statistical differences between certain features of spyware execution traces and benign traces. For example, the gray spots (*i.e.,* some specific instructions either previous to or posterior to the `Read`) in Fig. 3 may serve as such features. It is worth noting that the send behaviors in Fig. 3 cannot be directly tracked through execution traces without taint propagation.

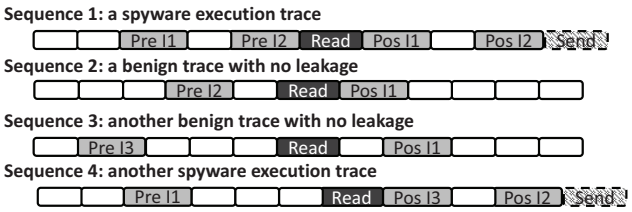


Fig. 3: Four exemplary execution sequences related to different privileged data reading behaviors

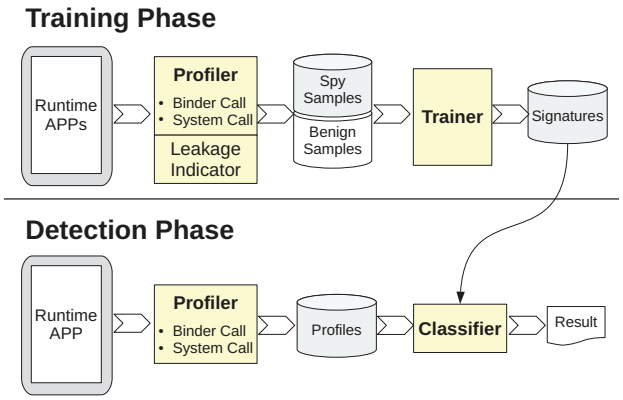


Fig. 4: Overall framework of SpyAware

Finally, if the hypothesis is true, we can solve the data flow analysis problem through a heuristic approach that only requires analysing the program execution trace, whose data are much easier obtainable in real-world scenarios.

IV. Our Approach

SpyAware employs binder calls and system calls to profile app dynamic executions, and discriminates privacy leakage behaviors accordingly. As we have discussed in Section II, a privacy leakage behavior usually starts with a `read`, which can be detected through binder calls. The challenging part is to determine whether the `read` behavior will eventually cause a leakage. Therefore, we first detect suspicious execution profiles which contain the `read` behaviors, and then evaluate whether each suspicious profile should be classified as a spyware execution.

The evaluation process involves two phases: a training phase and a detection phase. In the training phase, a number of labelled profiles (*i.e.,* the execution trace segment together with whether it indicates a leakage) are processed so as to learn the leakage signatures. In the detection phase, the runtime suspicious profiles captured during execution, which may or may not involve leakages, are labelled according to the signatures trained in the training phase. In this way, our approach can detect runtime leakage behaviors. Fig. 4 illustrates the overall framework.

Our approach is composed of two major components: 1) a portable profiler to instrument app dynamic executions, and to obtain the corresponding profiles of each UI event. 2) a pattern recognition algorithm to extract features from the profile samples, and to classify them. We discuss the details of each part in what follows.

A. App Instrumentation

Currently, the most widely adopted instrumentation approach on Android or other linux kernel-based OS is to trace system calls [7], which has been proved very effective in detecting some malware families. However, system calls are too low-level, and contain little Android-specific semantic information. We can even hardly know whether the privileged data have been accessed by using system calls only. Therefore, system calls are not enough for detecting spyware behaviors. Besides system call, a new light-weight approach specially tailored for Android OS is needed. Our approach is to leverage the characteristics of Android binder-based IPC and trace binder calls accordingly, which contain rich semantic information. Also, the profiler should be pluggable, and easy to use in the wild. We discuss the detailed design as follows.

1) Profile Binder Transactions: Binder-based IPC is a unique feature on Android, and is extensively used during an app lifecycle. Binder is a virtual device that allows processes to register with unique name identifiers (mapping to handlers) for calling each other. Each process communicates with the binder through a native library, *i.e.,* `libbinder.so`. With the methods provided by `libbinder.so`, a binder call is wrapped into `parcel` first, and is eventually sent via `ioctl` (*i.e.,* a function of `libc.so`) in `binder_write_read` structure. Fig. 5 illustrates the details of such a data structure. The content of binder calls can be interpreted by properly decoding the data. Fig. 6 is an example of the data after decoding.

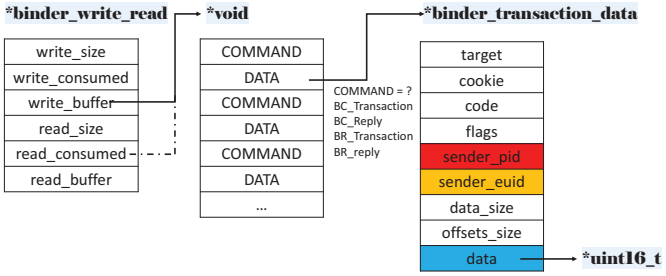


Fig. 5: The data structure of a binder call

To instrument the binder call of a target process, we can hook the `ioctl` of `libbinder.so`. We inject a dynamic library into the maps file of the target application process. The library contains a customized `ioctl` function. We modify the address of `ioctl` within the GOT (Global Offset Table) of `libbinder.so`, so that all the related `ioctl` function calls can be redirected to our customized `ioctl`. An interpreting process is performed by decoding the parameters of `ioctl` according to the data structure shown in Fig. 5. The calls are redirected to the real `ioctl` afterwards. Note that, in this way, we may also intercept `ioctl` calls for other I/O devices, which can be filtered out easily by interpreting their parameter values (*i.e.,* the value of the `request` parameter should be `BINDER_WRITE_READ`). Using the such an instrumentation method, we can trace all the binder calls.

#	Command	Data
1	BC_TRANSACTION	****android.app.IActivityManager**********%*com.sec.multiwindow.MW_TOUCH_DETECTED**********`*********mw_x*****e*****mw_action*********mw_y*********
2	BR_REPLY	****
3	BC_TRANSACTION	****android.content.IContentProvider****GET_system****sound_effects_enabled***
4	BR_REPLY	****
5	BC_TRANSACTION	**"*android.gui.DisplayEventConnection**
6	BR_REPLY	**$*********value*****0*
7	BC_TRANSACTION	****android.app.IActivityManager**********com.android.contacts****
8	BR_REPLY	**$*********value*****0*
9	BC_TRANSACTION	****android.content.IContentProvider****'*content://com.android.contacts/contacts*****_id**********************
10	BR_REPLY	****0*com.android.providers.contacts.ContactsProvider2****com.android.providers.contacts**********com.android.providers.contacts**********com.android.providers.contacts*****android.process.acore**********#*/system/app/SecContactsProvider.apk*#*/system/app/SecContactsProvider.apk*-*/data/data/com.android.providers.contacts/lib*****!*/system/framework/sec_feature.jar**/data/user/0/com.android.providers.contacts**********android.process.acore*********contacts;com.android.contacts***android.permission.READ_CONTACTS**!*android.permission.WRITE_CONTACTS*********.**********
11	BR_REPLY	**********_id*********}********************************f***

Fig. 6: An example of decoded data when accessing contact list. We use "*" to represent data which are binary codes (*i.e.,* not human-readable texts)

To obtain useful information from BINDER_WRITE_READ structured data as shown in Fig. 5, we decode the data field within binder_transaction_data. Fig. 6 shows an example of such data. We observe that the data along with BC_TRANSACTION (*i.e.,* a binder command type) usually start with a name identifier (*e.g.,* 'android.app.IActivityManager' of the first sequence), and are followed by the corresponding parameters. In general, binder transaction data are very complex. For example, some data in Fig. 6 have '*' (*i.e.,* binary code) and very specific details which may not be repeatable. A profile would not be helpful for spyware behavior detection if it is not repeatable. Therefore, we only reserve the name identifiers of BC_TRANSACTION data to represent a binder call, which should follow the Android framework standard. Note that there are four binder command types defined by Android, but during our experiment, we find only two of them are used, *i.e.,* BC_TRANSACTION and BR_REPLY. For the data following BC_REPLY, we discard them directly since most of them are very specific and data dependent. Fig. 7 shows an example of traces after stemming.

#	Command	Data
1	BC_TRANSACTION	android.app.IActivityManager
2	BC_TRANSACTION	android.content.IContentProvider
3	BC_TRANSACTION	android.gui.DisplayEventConnection
4	BC_TRANSACTION	android.app.IActivityManager
5	BC_TRANSACTION	android.content.IContentProvider

Fig. 7: Example of stemmed binder calls for Fig. 6

By analyzing the original BC_TRANSACTION data, we can identify private data access behaviors. For example, a URI 'content://sms' indicates that the app is reading SMS via content provider. We observe that most of the binder-based private data readings can be captured similarly. Fig. 8 defines several signatures which indicate possible private data readings.

Signature in Binder Call	Data Type
android.os.IServiceManager****iphonesubinfo	IMEI, ICCID
android.os.IServiceManager****location	Location
android.location.ILocationManager***gps	Location
android.location.ILocationManager***network	Location
android.location.ILocationManager***passive	Location
content://call_log/	Call History
content://com.android.contacts/	Contact
android.content.IContentProvider + com.android.contacts	Contact
content://sms/	SMS
content://browser/bookmarks	Browser History
android.gui.Sensor	Accelerometer
android.hardware.Camera	Camera

Fig. 8: Example signatures in binder calls which indicate possible private data readings

2) Profile System Calls: Strace is a standard system call tracing tool on Android platform, which we employ directly to instrument system calls. Since system calls contain little Android semantic information, and their parameters are usually very specific and data-dependent, we stem the parameters and reserve only their function names in the profile.

3) Separation of Traces: During the program executions, the profiles of successive operations are concatenated with each other. Since the objective of our work is to find the relationships between UI operations and data leakage behaviors, the profiles should be separated according to UI events. On Android platforms, UI events can be captured directly via reading the input data to the devices under /dev. Android also provides a standard tool, *i.e.,* getevent, to retrieve these UI events in a simple manner. In our work, we manually write such a tool based on getevent.

B. Feature Extraction

In this section, we define several features that can be applied to discriminate leakage profiles from other suspicious ones. Since our work aims at detecting privacy leakage during runtime, the feature extraction and classification methods should be performed online eventually, which implies that the features should be easily extracted and compared. Hence, we define all the features as binaries, *i.e.,* the values should be 0 or 1.

1) Read on Launch: According to the previous case study, many privacy leakage behaviors happen during app launch time, which implies if an app reads private data during launch time, it is very likely that the data would be leaked.

2) Features of Binder Calls: We observe that some method invocations are related to the spyware behaviors, *e.g.,* a spyware usually calls the network related classes and methods. These invocations usually involve inter-process communications and can be captured in binder calls. Based

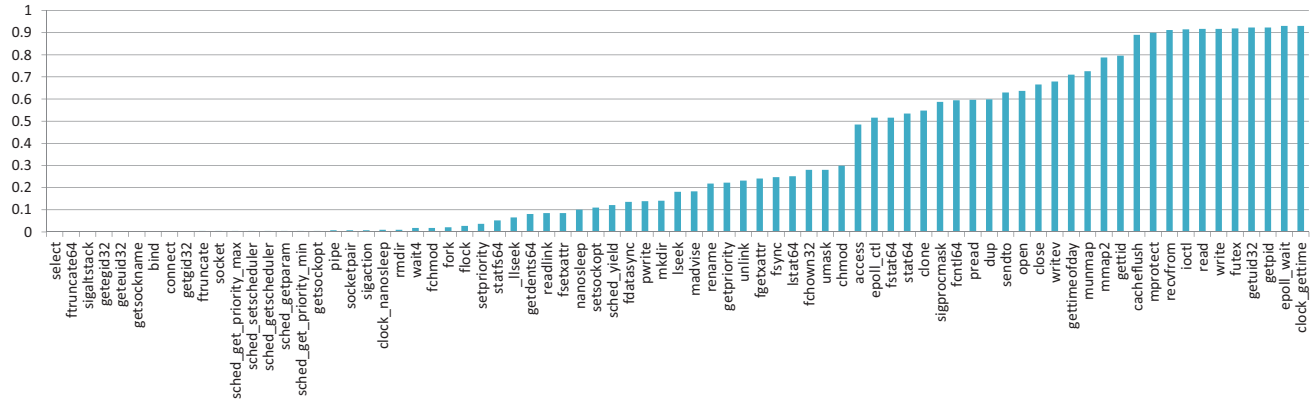


Fig. 9: The distribution of system call document frequencies

on the occurrence and position of the invocation within a profile, we define `happen before` and `happen after` features for each invocation of interest. If the invocation occurs before the `read`, the `happen before` feature is assigned 1, otherwise it is 0. Similarly, if the invocation occurs after the `read`, the `happen after` feature is assigned 1, otherwise it is 0. We discuss the invocations of our interest as follows:

IActivityManager: We have observed that some leakages are not intended by users, but happen automatically when users start a new activity. `android.app.IActivityManager` responses for the activity start and lifecycle management, and hence is of our interest.

IApplicationThread: The thread lifecycle management of Android activity and service is generally implemented with an `android.app.IApplicationThread` class, which would be invoked when operating (*e.g.,* creating or destroying) on the thread of the activity.

IConnectivityManager: Before accessing the Internet, apps usually check the current network connection status of the smartphone, *e.g.,* Is the mobile phone connected to the Internet or offline? Is it a 3G or WiFi connection? Such checking requires calling `android.net.IConnectivityManager`.

IWifiManager: `android.net.wifi.IWifiManager` is another call that may reflect some Internet behaviors, especially detailed WiFi connection information.

IMessenger: Network and some `read` behaviours usually work in a blocking mode. For safety reasons, Android developers usually assign such kind of tasks in another thread. `Messenger` is a common method to pass event or values between threads. Such communications call `android.os.IMessenger`.

IInputMethodManager: Some leakages happen when an app is querying the server, which carries data from the client side. To improve the response time and user experience, the query may perform periodically in the background when a user inputs the query data. The input usually involves the interaction with `com.android...view.IInputMethodManager`.

3) Features of System Calls: A system call usually starts with a function name, followed by its parameters and the return value. System calls are very low-level, whereas even a simple operation may trigger hundreds of system calls. Processing the system calls would be quite time-consuming and not applicable for online usage. For simplicity, we only keep the function name and remove the other parts. In this way, the profile can be represented with standard system call function names from a vocabulary. We observe that several system calls (*e.g.,* `ioctl` and `epoll_wait`) have very high occurrence frequencies. It is likely that these system calls contribute little information to distinguish the spyware behaviors.

We therefore adopt the idea of document frequency (*i.e.,* how many documents contain a term) to select system calls that are more informative. Taking system calls for each UI event as a document, we calculate the ratio of documents which contain a specified system call. The resulting document frequencies are shown in Fig. 9.

From the figure, we observe that some system calls have very small document frequencies. If we choose them as features, the feature vector matrix related to these features would be very sparse, *i.e.,* most of them are 0. They will help little for the classification purpose. Therefore, besides being informative, we require the system call to have adequate document frequency. Empirically, we choose the system calls with document frequency between 0.06 and 0.22. There are 13 system calls located in this range: `_llseek`, `getdents64`, `readlink`, `fsetattr`, `nanosleep`, `setsockopt`, `sched_yield`, `fdatasync`, `pwrite`, `mkdir`, `lseek`, `madvise`, `rename`.

Note that we may also employ term frequency/inverse document frequency (tf-idf) [20], or choose system calls within other document frequency regions, which might be more discriminative. But the idea is similar. A comparison study is performed in Section V.

C. Classifier

Our framework can seamlessly incorporate most of the popular classifiers for supervised learning. In this work, we investigate on applying two classifiers: Support Vector Machine (SVM) with Radial Basis Function (RBF) kernel and Naive Bayes Classifier (NBC) [5].

We adopt SVM with RBF kernel as our classifier, since it has long been proven successful in many classification applications. SVM can find a margin that best separates the classes of vectors. With such a margin, the classifier can classify an unlabelled sample of the feature vector according to which side of the margin it is located.

Naive Bayes Classifier, on the other hand, is a probability-based classier. It runs fast, and is thus more suitable for our scenario, which requires making classification decisions within a short time. In order to classify a new unlabelled sample of the feature vector, NBC calculates the probability of each class to generate the feature vector, and multiply it with the prior probability of each class. The label of the class with the greatest probability is assigned to the sample. Details can be found in [5].

D. Detection Algorithm

Now we introduce how our framework can be employed for online spyware behavior detection. An online detection system differs from offline analysis in that it requires timely response yet suffers the limitation of computational resources. We have already taken such requirements into consideration in our feature design step. Besides, we also consider how to extract these features efficiently. Our detection algorithm applied for online spyware detection is shown in Algorithm 1. Specially, the whole profile processing step, including identifying suspicious event and feature extraction, requires parsing the raw profile stream (*i.e.,* a execution trace segment) only once, *i.e.,* in a linear complexity.

In Algorithm 1, we assume that the model of spyware behavior ($spySig$), the signature array which indicates particular private data reading behaviors ($readSig$), the array indicating specific binder call features ($binderFeatureSig$), and the array indicating system call features ($sysFeatureSig$) are given as input. Our algorithm processes the runtime profile stream ($profileStream$) line by line. If a temporal line starts with a `BINDER` tag, which means it is a binder call, the algorithm extracts the binder feature by comparing the line with the predefined $binderFeatureSig$, or discriminates whether it contains a read signature by comparing with the predefined $readSig$. If it starts with a `SYSCALL` tag, the algorithm extracts system call features accordingly. We use a bit sequence to represent the feature vector, and each bit responses for one feature. If a feature can be extracted, the corresponding bit is set to one, otherwise zero. Finally, if the line starts with a `UI_EVENT` tag, which indicates the end of a profile, the algorithm justifies whether the previous profile should be classified as a spyware execution or not.

```
Data: profileStream, spySig
Data: readSig, sysFeatureSig, binderFeatureSig
readType ← −1;
while TREU do
    tmpStr ← Read(profileStream);
    if tmpStr.startwith(BINDER) then
        f ← ExtractBinderFeature(tmpStr);
        // Extract features by comparing
        with the readSig and binderFeatureSig
        if f > 0 then
            // A feature is extracted
            insFeature.setbit(f);
            // set to one
        end
        if f < 0 then
            // A read behavior is detected
            readType ← −f ;
        end
    end
    if tmpStr.startwith(SYSCALL) then
        f ← ExtractSyscallFeature(tmpStr);
        // Extract features by comparing
        with the sysFeatureSig
        if f > 0 then
            insFeature.setbit(f);
            // set to one
        end
    end
    if tmpStr.startwith(UI_EVENT) then
        // The end of a profile
        if readType > 0 then
            isSpy ← Classify(insFeature, readType);
            // Classify based on the spySig
            if isSpy then
                SendNotification();
            end
        end
        insFeature.clear();
        // Set all bits to zero
        readType ← −1 ;
    end
end
```

Algorithm 1: Detection Algorithm

V. Performance Study

A. Experimental Setup

In this section, we examine whether the proposed SpyAware framework is effective in finding the correlation between the spyware behaviors and execution traces. Specifically, we focus on two questions: 1) Are our features effective? 2) What accuracy can our approach achieve? To this end, we conduct an experiment with 100 popular apps downloaded from Google Play. We manually run each application as a normal user on TaintDroid for Android version 4.3 installed on Galaxy Nexus[2]. Meanwhile, our profiler runs in the background. In this way, we collect our raw data, *i.e.,* a large set of binder calls, system calls, UI events, together with the leakage indications for each UI event. We do not adopt automated testing (*e.g.,* using Monkey) because such tools cannot handle user registration and login issues, and hence cannot trigger some spyware behaviors effectively. Among these apps, five have compatibility issues with our tool: two of them do not support `ptrace`, and the others will automatically crash when being injected with our payload. The 3 crashing apps are `com.tao.taobao` and `com.alibaba.aliexpresshd` from Alibaba, and `com.snapchat.android` from Snapchat. We think they use some program integrity protection mechanisms. However, our tool works well with all the other apps.

Since the detection of suspicious profiles with `read` is rule-based, it is considered accurate. Our major challenge then is to examine the effectiveness of recognizing spyware

[2]TaintDroid does not support Android version 4.4 and later so far.

behaviors from suspicious ones. To this end, we filter out all the non-suspicious profiles (textiti.e., without a `read` detected in binder calls) and keep the suspicious ones. In this way, we find 56 apps generate 347 suspicious profiles for device id leakage, 139 of which are spyware behaviors and the others are benign. We also find 51 apps generate 171 suspicious profiles for location leakage, 51 of which are spyware behaviors and the others are benign. In what follows, we evaluate the capability of our approach in detecting these two kinds of data leaking. It is worth noting that the profiles reading device id and location are very common, while reading other data types are relatively less. Conducting experiments on other data types requires more experimental apps. However, we think the evaluation results on these two data types are also meaningful representatives for others.

B. Feature Evaluation

In Section IV-B, we have proposed binder call based and system call based features. However, are both of them effective? Which one works better? To answer these questions, we extract the binder call based features and system call based features separately for all the suspicious profiles.

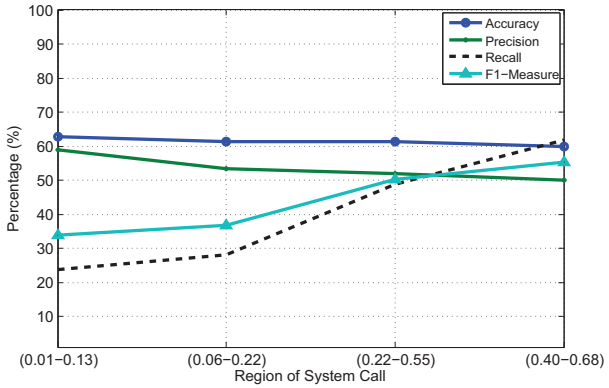
We evaluate the feature effectiveness using Naive Bayes Classifier with 10-fold cross-validation. In Naive Bayes Classifier, the classifier is firstly trained with a set of samples. Then, given a new unlabelled sample, the probability to be either a spyware behavior ($Prob_{spy}$) or not ($Prob_{nonspy}$) can be calculated respectively. If $Prob_{spy} > Prob_{nonspy}$, the sample can be classified as spyware, and vice versa. Our feature evaluation result is shown in Table I.

TABLE I: An effectiveness comparison study of system call based features (document frequency between 0.06 and 0.22) with binder call based features

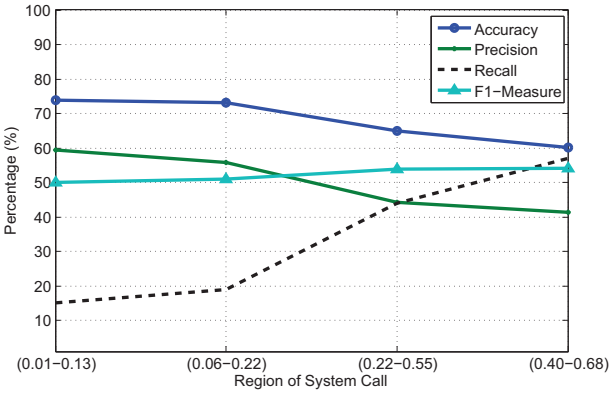
		Positive	Negative	Total	Accuracy
Device ID					
Binder call	True	74	145	219	63.1%
	False	63	65	128	
System call	True	68	145	213	61.4%
	False	63	71	134	
Location					
Binder call	True	25	94	119	69.6%
	False	26	26	52	
System call	True	35	76	111	64.9%
	False	44	16	60	

From Table I, we observe that binder call based features perform better than system call based features. However, this may not hold for other possible regions of system calls. In the previous feature extraction step, we have empirically chosen 13 system calls with document frequency between 0.06 and 0.22 as features. There are also other possible choices. We then evaluate the features of other possible choices. We conduct three more comparison experiments with different regions of system calls: the first one is to choose the system calls with document frequency between 0.01 and 0.13, the second one between 0.22 and 0.55, and the third one between 0.40 and 0.68. To be comparable, all three regions have 13 system calls.

To better visualize the performance variance of different system call regions, besides accuracy, we adopt 3 other



(a) Performance on device id leakage detection



(b) Performance on location leakage detection

Fig. 10: A performance comparison of different system call based features from 4 regions

metrics: precision (true positives over all positives), recall (true positives over true positives and false negatives) and F1-measure (the harmonic mean of precision and recall). Our results in Fig. 10 show that features of system calls with low document frequency perform better in achieving high accuracy. However, their recall is relatively low, thus affecting its overall performance in F1-measure.

C. Overall Performance

We now evaluate how effective our approach can achieve with both binder call based features and system call based features. Although the previous evaluation result shows that the system call based features within region 0.01-0.13 perform best in accuracy, it is not good in F1-measure. To avoid bias, we adopt the features of system calls within the region of 0.06-0.22.

We first use Naive Bayes classifier to evaluate the performance. The result is shown in Table II. To demonstrate that our approach is effective, we should compare the results with what naive guesses can achieve. Suppose a guesser has pre-knowledge that 139 out of 347 suspicious samples for device id leakage are spyware, he can simply guess all the unlabelled instances as benign to get the best performance in accuracy, which is 59.6% (1-139/347). Similar, the best accuracy that a naive guesser can achieve for location leakage

is 70.2% (1-51/171). We observe that the accuracies in our result are better than the naive guesses. Moreover, in order to achieve the best performance in accuracy, the naive guesses have sacrificed the recalls, which are 0% for both device id and location. Comparatively, our recalls are 43.9% (61/139) and 49% (25/51). Therefore, the correlation between the data flow of privacy leakage behaviors and the specific features of execution traces can be justified.

TABLE II: Detection capability with Naive Bayes Classifier

		Positive	Negative	Total	Accuracy
Device ID	True	61	162	223	64.2%
	False	46	78	124	
Location	True	25	103	128	74.9%
	False	17	26	43	

Naive Bayes Classifier is relatively weak in achieving good performance, because it does not consider the relationship among features. Next, we try a more sophisticated classier, *i.e.,* SVM with RBF kernel. Since the performance of such a classifier heavily relies on the parameter settings, we manually tune the SVM parameters to get an optimal solution, *i.e.,* we use *-c 3 -g 0.2 -t 1 -d 10* in the experiment. Our experimental result is shown in Table III.

TABLE III: Detection capability with SVM

		Positive	Negative	Total	Accuracy
Device ID	True	59	175	234	67.4%
	False	33	80	113	
Location	True	21	113	134	78.4%
	False	7	30	37	

From Table III, the performance gets slightly improved. The detection accuracy for location leakage even raises up to 78.4%, which is far better than the accuracy of the naive guess. However, it still cannot approach 100%. Besides, the false negative number in the result is relatively high, which implies a low recall. We may investigate more effective features to improve the performance in the future.

In the previous evaluations, we randomly separate all the suspicious profiles into 10 folds. We may take advantage of using the profiles of a special app to classify other profiles from the app itself. Now we evaluate how effective our approach can achieve when detecting spyware behaviors without the previous knowledge of the particular app, *i.e.,* the cross-app detection capability. We also adopt 10-fold cross-validation and the same SVM parameter setting. Our experimental result is shown Table IV. The performance is slightly worse than Table II and Table III, but still much better than the naive guesses.

As a short conclusion, the experimental results have initially verified the effectiveness of our approach. Nevertheless, future research on more effective approaches is still needed to make the approach more practical.

D. Discussion

Our approach targets on examining the correlation between the data flow of privacy leakage and some features of program executions. Even though the accuracies achieved in the benchmark are not yet ready for real world deployment, they have significant statistical differences in comparison with naive guesses, and the correlation has been demonstrated. In the evaluation experiment, we have shown that with only hundreds of spy samples from tens of apps, we can predict the spyware behaviors in promising improvements on accuracy. Note that with millions of installation in mobile apps, even 1% of accuracy improvement in privacy protection represents significant reputational and commercial benefits. Besides, our approach is also effective for detection with cross-app signatures and thus would not suffer the scalability issue. However, there is still room to improve the performance, which indicates a need for further research. One major direction lies in our feature extraction approach which only includes binary features, and consequently may sacrifice the effectiveness as a trade-off for the efficiency. If employing some complex features, the model performance can be further improved.

TABLE IV: Cross-app detection capability with Naive Bayes Classifier and SVM

		Positive	Negative	Total	Accuracy
Device ID (NBC)	True	49	168	217	62.5%
	False	40	90	130	
Location (NBC)	True	25	103	128	74.9%
	False	17	26	43	
Device ID (SVM)	True	53	175	228	65.7%
	False	33	86	119	
Location (SVM)	True	18	113	131	76.6%
	False	7	33	40	

VI. Related Work

The privacy and security issues of smartphones incurred by third-party apps have received extensive attention. A number of approaches and tools have been proposed to combat malicious code. Existing work in this area mainly focuses on the weakness of permission-based security mechanism (*e.g.,* [11, 13, 29]) and malware issues (*e.g.,* [7, 8, 15, 26, 27, 33, 34]). In contrast, the privacy issue, which we focus in this work, is an area that attracts less effort. The work on Android privacy involves two major areas: 1) tackling the privacy issues through improving the Android security mechanism. 2) analyzing app spy behaviors with static analysis or dynamic analysis approach. We discuss each of them as follows.

Several investigations that propose to improve Android privacy protection mechanism can be found in [4, 6, 21]. They argue that Android's permission system is one of the root causes for privacy problems. Felt *et al.* first notice that Android permission mechanism may cause some security issues [11, 13]. Kelley *et al.* [17] has done another interesting work on investigating how permission and privacy declaration affect user application selection decisions. They find that users tend to choose applications with less permissions. Nauman *et al.* [21] notice Android adopts a pre-installation permission grant mechanism and users have no fine-grained permission choices (either accept all or exit installation process) during installation. They present Apex, which is a policy enforcement framework for Android that allows a user to selectively grant permissions to applications. Benats *et al.* [4] consider Android

has permission redelegation or escalation problems and has no support for checking permission conflicts about privacy. They propose an extension to the traditional P-RBAC model that can mitigate such weaknesses. To provide Android OS with enhanced security features of instantiating different security solutions, Bugiel *et al* propose FlaskDroid [6]. FlaskDroid provides mandatory access control simultaneously on both the Android middleware and kernel layer.

Static analysis approach is an effective approach in analyzing privacy issues of applications adopted by several work (*e.g.,* [14, 18, 19, 25]). The advantage of a static analysis approach is that it can analyze a large number of applications in a short period of time, which is very efficient. However, since popular commercial applications usually have code protection mechanism and use native code, such mechanism is usually not applicable for these apps. Moreover, a pure static analysis result is more useful for application markets, instead of smartphone users. FlowDroid [3] is such a work that use static analysis approach to detect privacy leakage, it is very effective in tracking some Android-specific data flow and detecting leakages. However, it still suffers the problem of inaccuracies because its code analysis is based on the reverse engineering techniques instead of the source code directly. Moreover, FlowDroid can only be used on PC, and not applicable on smartphones. PiOS [9] is another work similar to FlowDroid but for iOS applications. Several other investigations (*e.g.,* [25, 32]) adopt a hybrid approach that use the static analysis results to assist a dynamic analysis process. Noticing Android permission mechanism does not convey meaningful information on how a user's privacy might be impacted by using an application, Rosen *et al.* [25] propose to generate high-level privacy-related profiles based on a static analysis approach, such profiles can also be used on smartphones for runtime leakage detection. But they haven't considered whether the data would be leaked after being read. Yang *et al.* [32] argue that the transmission of sensitive data in itself does not necessarily indicate privacy leakage. However, if the transmission occurs without user's attention, it is more likely to be a leakage. They propose AppIntent, which applies a guided symbolic execution approach to detect the leakage. However, AppIntent requires the source code of application, which is impractical for the end users.

To provide users with adequate visibility into how third-party applications use their private data, Enck *et al.* propose TaintDroid [10]. TaintDroid implements a novel dynamic taint analysis approach on Android, which includes four levels of taint tracking: message-level, variable-level, method level and file-level. Such a multiple granularity approach has been proved effective and very efficient in achieving only 14% CPU overhead. However, TaintDroid requires user to recompile and flash the operation system, which is impractical for ordinary users. Qian *et al.* [22] also propose using dynamic taint analysis approach to detect privacy leakage, and their work extends the detection capability to support native library comparing with TaintDroid, but it incurs more overhead. To use the dynamic taint analysis feature to automatically analyse apps, Rastogi developed AppsPlayground that automates the analysis of smartphone applications [23]. Xu *et al.* [30] find out that most research on enhancing the platform's security and privacy controls requires extensive modification to the operating system, which has significant usability challenges. They propose Aurasium, an automated repacking tool that attach user-level sandboxing code to arbitrary original applications. Such an approach can eliminate the needs to modify Android OS. However, since Aurasium does not trace the operations on sensitive data and transmission behaviors, its privacy leakage detection capability is very limited. Note that there are also other investigations that use dynamic analysis approach to detect malware, *e.g.,* [7, 24]. Because their focus is not privacy leakage, we do not discuss them in detail. Our work is different from the afore mentioned work in that we focus on combating privacy leakage issues, and it is the first attempt to solve this problem with dynamic signature-based detection approach.

VII. Conclusion

This paper serves as the first attempt to investigate the correlation between the data flow of privacy leakage and app execution traces, which may be as a viable means for smartphone privacy leakage detection. To this end, we have proposed our SpyAware framework, incorporating a set of methods targeting on obtaining the execution traces and extracting effective features so as to discriminate the spyware execution during app runtime. Our approach relies on no app-specific information and the features are hence applicable for cross-app detection. Specially, our design takes the online usage into consideration, which requires timely response with limited computational resource: 1) our profiler is portable and efficient, as it leverages the characteristics of Android Binder-based IPC which contains rich semantic information, and we only need to inspect one method (*i.e.,* `ioctl`) to get the information. 2) we only employ binary features and the feature extraction algorithm is efficient in linear complexity.

We have further presented our experiences on applying SpyAware over 100 popular Android apps. Experimental results have shown that our approach can achieve promising results, but there is still room for improvement towards its practical application. To conclude, we believe this initial work sheds light on future research towards combating smartphone privacy leakage issues.

A number of tasks have been identified as follow-up research. First, we can investigate on how to improve the accuracy by trying different features and algorithms. Secondly, in real world, there are various types of Android smartphones with customized OS; therefore, the effectiveness of signatures across different versions of Android OS and platforms should also be evaluated. Finally, since the leakage can happen frequently during user's ordinary usage, a user-friendly notification mechanism should be designed.

Acknowledgements

This work was supported by the National Basic Research Program of China (973 Project No. 2014CB347701), the Key Project of National Natural Science Foundation of China (Project No. 61332010), the Research Grants Council of Hong Kong (Project No. CUHK 14205214), and Microsoft Research Asia Grant in Big Data Research (Project No. FY13-RES-SPONSOR-036).

REFERENCES

[1] LBE. *http://www.lbesec.com*.

[2] Qihoo360. *http://shouji.360.cn/index.html*.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[4] G. Benats, A. Bandara, Y. Yu, J. Colin, and B. Nuseibeh. Primandroid: privacy policy modelling and analysis for android applications. In *Policies for Distributed Systems and Networks, 2011 IEEE International Symposium on*, pages 129–132. IEEE, 2011.

[5] C. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[6] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *22nd USENIX Security Symposium*, 2013.

[7] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 15–26. ACM, 2011.

[8] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24. ACM, 2013.

[9] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation*, volume 10, pages 1–6, 2010.

[11] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638. ACM, 2011.

[12] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension. Technical report, and behavior. Tech. Rep. UCB/EECS-2012-26, UC Berkeley, 2012.

[13] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[14] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.

[15] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile systems, Applications, and Services*, pages 281–294. ACM, 2012.

[16] L. Hutton, T. Henderson, and A. Kapadia. Short paper:"here i am, now pay me!": Privacy concerns in incentivised location-sharing systems. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, pages 81–86. ACM, 2014.

[17] P. G. Kelley, L. F. Cranor, and N. Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the 2013 ACM Annual Conference on Human Factors in Computing Systems*, pages 3393–3402. ACM, 2013.

[18] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. Scandal: Static analyzer for detecting privacy leaks in android applications. 2012.

[19] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[20] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*. Cambridge university press Cambridge, 2008.

[21] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[22] C. Qian, X. Luo, Y. Shao, and A. T. Chan. On tracking information flows through jni in android applications. In *Dependable Systems and Networks, 2014 44th Annual IEEE/IFIP International Conference on*, pages 180–191. IEEE, 2014.

[23] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[24] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[25] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and Application Security and Privacy*, pages 221–232. ACM, 2013.

[26] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *Intelligence and Security Informatics Conference, 2012 European*, pages 141–147. IEEE, 2012.

[27] A. Schmidt, R. Bye, H. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. IEEE, 2009.

[28] J. Tan, U. Drolia, R. Martins, R. Gandhi, and P. Narasimhan. Short paper: Chips: content-based heuristics for improving photo privacy for smartphones. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, pages 213–218. ACM, 2014.

[29] C. Thompson, M. Johnson, S. Egelman, D. Wagner, and J. King. When it's better to ask forgiveness than get permission: attribution mechanisms for smartphone resources. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, page 1. ACM, 2013.

[30] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security Symposium*, pages 27–27. USENIX Association, 2012.

[31] W. Xu, F. Zhang, and S. Zhu. Permlyzer: Analyzing permission usage in android applications. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 400–410. IEEE, 2013.

[32] 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 *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[33] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pages 611–622. ACM, 2013.

[34] 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 *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, pages 5–8, 2012.