Permission Use Analysis for Vetting Undesirable Behaviors in Android Apps

Yuan Zhang, Min Yang, Zhemin Yang, Guofei Gu, Peng Ning, and Binyu Zang

***Abstract*— The android platform adopts permissions to protect sensitive resources from untrusted apps. However, after permissions are granted by users at install time, apps could use these permissions (sensitive resources) with no further restrictions. Thus, recent years have witnessed the explosion of undesirable behaviors in Android apps. An important part in the defense is the accurate analysis of Android apps. However, traditional syscall-based analysis techniques are not well-suited for Android, because they could not capture critical interactions between the application and the Android system. This paper presents VetDroid, a dynamic analysis platform for generally analyzing sensitive behaviors in Android apps from a novel permission use perspective. VetDroid proposes a systematic permission use analysis technique to effectively construct permission use behaviors, i.e., how applications use permissions to access (sensitive) system resources, and how these acquired permission-sensitive resources are further utilized by the application. With permission use behaviors, security analysts can easily examine the internal sensitive behaviors of an app. Using real-world Android malware, we show that VetDroid can clearly reconstruct fine-grained malicious behaviors to ease malware analysis. We further apply VetDroid to 1249 top free apps in Google Play. VetDroid can assist in finding more information leaks than TaintDroid, a state-of-the-art technique. In addition, we show how we can use VetDroid to analyze fine-grained causes of information leaks that TaintDroid cannot reveal. Finally, we show that VetDroid can help to identify subtle vulnerabilities in some (top free) applications otherwise hard to detect.**

***Index Terms*— Android security, permission use analysis, vetting undesirable behaviors, android behavior representation.**

I. Introduction

SMARTPHONE platforms are becoming more and more popular these days [2]. To protect sensitive resources in the smartphones, permission-based isolation mechanism [3] is used by modern smartphone systems to prevent untrusted apps from unauthorized accesses. In Android, an app needs to explicitly request a set of permissions when it is installed. However, after permissions are granted to an app, there is no way to inspect and restrict how these permissions are used by the app to utilize sensitive resources. Unsurprisingly, Android has attracted a huge number of attacks. According to McAfee threat report of Q3 2012 [4], Android remains the largest target for mobile malware and the number almost doubled in Q4 2012. While these malware apps are clear examples containing undesirable behaviors, unfortunately even in supposedly benign apps, there could also be many hidden undesirable behaviors such as privacy invasion.

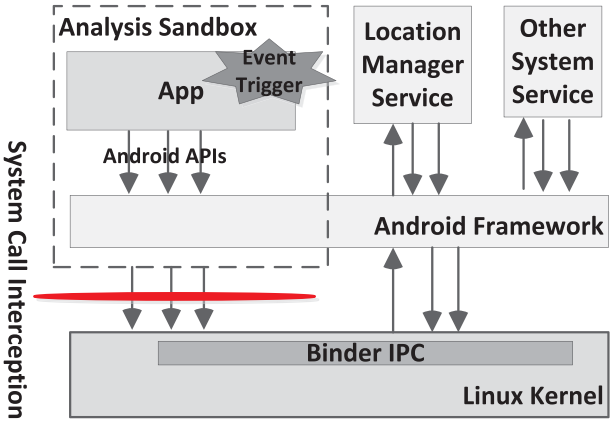


Fig. 1. Limitations of syscall-based solutions on Android platform.

An important part in the fight against these undesirable behaviors is the analysis of sensitive behaviors in Android apps. Traditional analysis techniques reconstruct program behaviors from collected program execution traces. A rich literature exists (see [5]–[11]) that focuses on solutions to construct effective behavior representations. All these research efforts have mostly used system calls to depict software behaviors because system calls capture the intrinsic characteristics of the interactions between an application and the underlying system. Previous studies differ from each other only in how to structure the set of system calls made by the applications [12]. However, previous work is not readily applicable due to the following unique features of Android.

A. Android Framework Managed Resources

As depicted in Figure 1, Android is an application framework on top of Linux kernel [13] where applications do not directly use system calls to access system resources. Instead, most system resources in Android are managed and protected by the Android framework, and the application-system

Manuscript received December 21, 2013; revised April 10, 2014 and July 23, 2014; accepted July 28, 2014. Date of publication August 12, 2014; date of current version October 3, 2014. This material is based upon work supported in part by the National Natural Science Foundation of China (61103078, 61300027), in part by the Science and Technology Commission of Shanghai Municipality (13511504402 and 13JC1400800), a joint program between China Ministry of Education and Intel numbered MOE-INTEL201202, and in part by the National Science Foundation under Grant CNS-0954096. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Dinei A. Florencio. *(Corresponding author: Min Yang.)*

Y. Zhang, M. Yang, Z. Yang, and B. Zang are with the School of Computer Science, Fudan University, Shanghai 201203, China (e-mail: yuanxzhang@fudan.edu.cn; m_yang@fudan.edu.cn; yangzhemin@fudan.edu.cn; byzang@fudan.edu.cn).

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

P. Ning is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206 USA (e-mail: pning@ncsu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2014.2347206

1556-6013 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

interactions occur at a higher semantic level (such as accessing contacts, call history) than system calls at the Linux Kernel level. Indeed, Android provides specific APIs for applications to access system resources and regulates the access rules. Using system calls to learn the interaction behaviors between applications and Android will lose a semantic view of accesses to Android resources, degrading the quality and precision of the reconstructed behaviors.

B. Binder Inter-Process Communication (IPC)

In Android, system services are provided in separated processes, with a convenient IPC mechanism (Binder) to facilitate the communication among system services and applications. Binder IPC is heavily used in Android and recommended in the design of applications. Figure 1 demonstrates the problems brought by the wide use of IPC to traditional syscall-level behavior reconstruction. First, traditional solutions would only intercept a lot of system calls used to interact with the Binder driver, hiding the real actions performed by the application. Second, the use of IPC in Android apps breaks the execution flow of an app into chains among multiple processes, making the evasion of traditional syscall-based behavior monitoring easier [14].

C. Event Triggers

Android employs an event trigger mechanism to notify interested applications when certain (hardware) events occur. In this model, for example, if an application wants to be notified when the phone's location changes, it just needs to register a callback for such an event. When Android sniffs a location change event from the location sensors, it notifies all the interested applications of the latest location by invoking their registered callbacks. Although the event notification is proceeded via Binder IPC (sycall), this asynchronous resource access model via system delivery is quite different from the synchronous application-request access model in three aspects. First, selecting privileged event notifications in a trace of syscalls requires ad-hoc Binder IPC dissecting. Second, intercepting event notification is far away from identifying the callbacks because application may have its specific logic in dispatching events to specific callbacks. Third, as showed in Figure 1, we could find that application registered callbacks are application code, so their executions cannot be captured with syscalls. As a result, traditional behavior reconstruction methods will lose such important behavior characteristics.

The above analysis indicates that a general method to analyze sensitive behaviors of Android apps is highly desired. Since Android does not use system calls as the main mechanism to isolate applications, system calls do not appear to be a good vehicle for representing behaviors. Considering the unique permission-based isolation mechanism in Android, we propose *Permission Use Analysis* to analyze sensitive behaviors in Android apps from a novel *permission use* perspective. We define a new concept, *permission use behavior*, which captures what and how permissions are used to access system resources, as well as how these resources are further utilized by the application internally. For example, assume an application requests both `ACCESS_FINE_LOCATION` and `INTERNET` permissions during the installation time. The *Permission Use Analysis* technique should track the *explicit* points where these two permissions are requested and also all the *implicit* points where the location and network resources are used inside the application. In this case, any point where two permissions are intertwined is of particular interest because it might indicate possible location leakage to the network.

In this paper, we design a dynamic analysis system called *VetDroid* to automatically perform permission use analysis on Android apps. However, it is non-trivial to construct an effective permission use analysis technique. *VetDroid* overcomes several key challenges to *completely* identify all permission-sensitive behaviors with *accurate* permission use information during the runtime. Such analysis is performed in two phases: first, *VetDroid* identifies all sensitive interactions between the Android system and apps with accurate permission use information by intercepting all invocations to Android APIs and sniffing exact permission check information from Android's permission enforcement system; second, based on the identified sensitive interactions, *VetDroid* tracks all potential sensitive behaviors inside the apps, by recognizing the exact delivery point in the application for each permission-sensitive resource and locating all the use points of these resources with permission-based tainting analysis. *VetDroid* also features a driver to enlarge the scope of the dynamic analysis to cover more application behaviors and a behavior profiler to generate behavior graphs with highlighted sensitive behaviors for analysts to examine.

To evaluate the effectiveness of permission use analysis and *VetDroid*, we first use *VetDroid* to analyze real-world Android malware. The results show that the permission use behaviors reconstructed by *VetDroid* can significantly ease the malware analysis. We further apply *VetDroid* to more than one thousand top free apps in Google Play Store. *VetDroid* finds more information leaks than the state-of-the-art leak detection system TaintDroid [1], and shows its capability to analyze the fine-grained incentives of information leaks among the apps. Furthermore, *VetDroid* even detects subtle *Account Hijack Vulnerability* in a top free Android app. The analysis overhead caused by *VetDroid* is reasonably low for an offline analysis tool.

This paper makes the following major contributions.

- We analyze the limitations of existing syscall-based behavior analysis methods when applied to Android platform and propose permission use analysis as a new perspective to analyze Android apps.
- We present a systematic framework to reconstruct permission use behaviors. Our automated solution is able to completely identify all possible permission use points with accurate permission information.
- We implement a prototype system, *VetDroid*, and evaluate its effectiveness in analyzing real-world Android apps. *VetDroid* not only greatly eases the analysis of malware behaviors, but also assists in identifying fine-grained causes for information leakages and even subtle vulnerabilities in benign Android apps otherwise hard to detect.

The rest of this paper is organized as follows. Section II introduces some background information about Android and the motivation for a new kind of behavior analysis technique. Section III presents our key permission use analysis technique and Section IV describes two auxiliary components necessary for an automated analysis system. After that, we present our evaluation results in Section V and discuss the paper in Section VI. Finally, Section VII presents the related work and Section VIII concludes the paper.

II. Problem Statement

A. Android Background

Android is the most popular mobile operating system today. To enhance the security, Android is designed to be a privilege-separated operating system, in which each application runs with a distinct system identity (Linux UID and GID). Android employs a quite efficient and convenient IPC mechanism, Binder, which is extensively used for interaction between applications as well as for application-OS interfaces. Binder is implemented as a kernel driver and user-level applications could just interact with it through standard system calls, e.g., open(), ioctl(). Binder is the key infrastructure of Android system and aggressively used to connect various parts of the system together.

To facilitate resource accessing from isolated applications and data sharing among applications and the system, Android designs a permission-based security mechanism [15]. Each application needs permissions to access system resources. These permissions are granted from users at install time. At runtime, each application is checked by Android before accessing sensitive resources. Any access to resources without granted permissions will be denied. The permission mechanism in Android is fine-grained [16] which is different from iOS [17]. In Android 4.2, there are 130 items of sensitive resources that are protected with permissions [18]. Limited by page space, we only introduce some basic knowledge about Android here and the conference version [19] of this paper includes more details.

B. Motivation

Existing work [5], [7]–[10] on behavior analysis has mostly used system calls to depict application's internal behaviors. However, previous work has problems when applied to Android platform due to Android's new security model. As explained in Section I, these problems make traditional solutions not well-suited for monitoring fine-grained Android behaviors such as accesses to Android managed resources, interactions with system services through Binder IPC, and responses to privileged system events. Based on syscall-level introspection, CopperDroid precisely reverses Android API invocations. Compared with existing syscall-based work, it could extract better Android-level semantics. However, only recognizing all interactions between the Android system and apps might not be enough for constructing an effective, fine-grained, internal behavior analysis. CopperDroid still has limitations in filtering out irrelevant interactions and examining fine-grained internal behaviors inside apps.

TABLE I
Capabilities Achieved by Existing Work When Compared With VetDroid

	Android level semantics	**Analyze General Behaviors**	**Analyze Internal Behaviors**	**Filter Irrelevant Behaviors**
Syscall-based	×	×	×	×
CopperDroid	√	√	×	×
TaintDroid, AppIntent	√	×	×	√
ProfileDroid	√	√	×	×
DroidScope	√	√	√	×
PEG	√	×	×	√
VetDroid	√	√	√	√

TaintDroid [1] alerts information leaks inside an Android app via dynamic taint tracking. AppIntent [20] redefines the privacy leakage as user-unintended sensitive data transmission and designs a new technique, event-space constraint symbolic execution, to distinguish intended and unintended transmission. However the two tools could neither analyze other kinds of undesirable behaviors such as stealthily sending SMS, nor examine the internal logic of sensitive behaviors. ProfileDroid [21] is a behavior profiling system for Android apps which is also not suitable for analyzing internal behavior logic. DroidScope [22] is an analysis platform designed for Android that extends traditional techniques to cover Java semantics. However, the problem of analyzing Android apps is not simple as how to capture behaviors from different language implementations. It is hard to conduct effective analysis without considering Android's specific security mechanism. Permission Event Graph [23], which represents the temporal order between Android events and permission requests, is proposed to characterize unintended sensitive behaviors. However, this technique could not capture the internal logic of permission usage, especially when multiple permissions are intertwined.

From the above short analysis of existing work, we find that they do not take full consideration of permission-based isolation mechanism in Android [3], which we believe to be important to understand behaviors of these applications. Thus, in this paper we propose *Permission Use Analysis* as a new and complementary aspect in analyzing Android apps. The proposed permission use analysis technique captures what and how permissions are used to access system resources, as well as how these resources are further utilized by the application internally. We build a dynamic analysis platform, named *VetDroid*, to automatically perform permission use analysis on Android apps. As presented in Table I, compared with existing work, *VetDroid* has the following capabilities in analyzing sensitive behaviors in Android apps.

- *Android-level Semantics*. To ease the analysis of Android apps, *VetDroid* extracts Android-level semantics during the analysis, such as Android APIs, permission usage.
- *Analyze Generic Sensitive Behaviors*. *VetDroid* supports analyzing generic, high-risk behaviors by effectively abstracting sensitive behaviors.

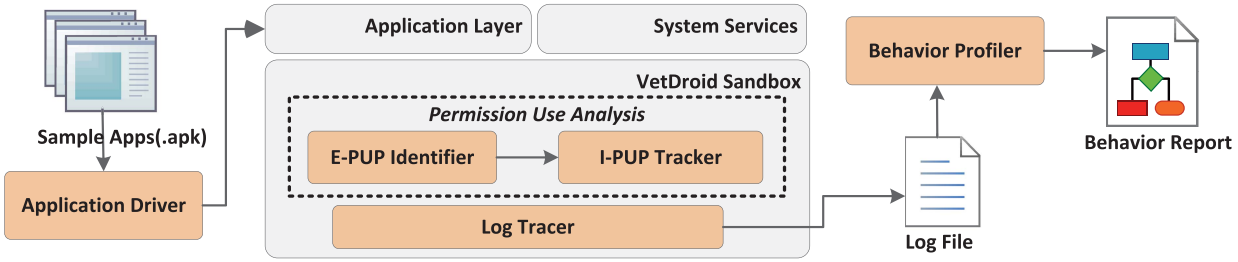


Fig. 2. Overview of *VetDroid* to analyze Android apps with permission use analysis.

- *Analyze Internal Behaviors*. The internal behaviors inside an Android app are crucial for analyzing Android apps. *VetDroid* not only identifies sensitive interactions between the Android system and apps, but also tracks sensitive internal behaviors.
- *Filter Irrelevant Behaviors*. Not all behaviors exposed by the app are valuable for security analysis. To highlight sensitive behaviors, *VetDroid* automatically filters irrelevant behaviors from the permission perspective.

C. VetDroid Overview

Figure 2 shows the overview design of *VetDroid*. Sample applications are first loaded into *Application Driver*, which automatically executes the application in our sandbox (details described in Section IV-A). During the execution, *Permission Use Analysis* module (details described in Section III) identifies all the permission use points and their relationships. These behaviors are recorded by *Log Tracer* with runtime information into a log file. The log file is offline processed by *Behavior Profiler* to automatically construct behavior representations (details described in Section IV-B). Next, this paper will detail the design of main components of *VetDroid* system.

III. Permission Use Analysis

The key challenge in our approach is on the effectiveness of *Permission Use Analysis*, i.e., how to *completely* identify all the permission use points with *accurate* permission information and *precisely* track their relationships. We define a new concept, *permission use behavior*, to represent the extracted behaviors in terms of permission use points. Specifically, we distinguish two kinds of permission use points in *permission use behavior*:

- *Explicit Permission Use Points (E-PUP)*, which represent sensitive interactions between the Android system and the app that *explicitly* cause permission checks.
- *Implicit Permission Use Points (I-PUP)*, which capture sensitive internal behaviors inside an app that *implicitly* utilize permission-related resources with application-specific logic.

Permission Use Analysis is proceeded in two phases. First, it identifies all sensitive application-system interactions that cause permission checks (aka *E-PUP*s), and marks these interactions with accurate permission use information. Second, it locates all the permission-sensitive resources that acquired from the system and track all the sensitive internal use points of these resources (aka *I-PUP*s). Since asynchronous resource

```
LocationManager locMan =
    (LocationManager)getSystemService(LOCATION_SERVICE);
locMan.requestLocationUpdates(LocationManager.GPS_PROVIDER,
        1000, 0, new SmsLocationListener());  ----------(1)
        Explicit Permission Use Point : ACCESS_FINE_LOCATION
public class SmsLocationListener() Implements LocationListener{
  public void onLocationChanged(Location location) {  ----------(2)
        Resource Delivery Point : ACCESS_FINE_LOCATION
    String message = "Lat: " + location.getLatitude()
          + " Lon: " + location.getLongitude();
    SmsManager smsMan = SmsManager.getDefault();
    smsMan.sendTextMessage("12345", null, message, null, null); -----(3)
        Explicit Permission Use Point : SEND_SMS
        Implicit Permission Use Point : ACCESS_FINE_LOCATION
  }
  public void onProviderDisabled(String provider) {}
  public void onProviderEnabled(String provider) {}
  public void onStatusChanged(String provider, int status, Bundle extras) {}
};
```

Fig. 3. An example about *E-PUP*s and *I-PUP*s.

delivery mechanism is commonly used in Android, *Permission Use Analysis* carefully takes this mechanism into account by precisely recognizing the resource delivery point, which means the first place in the app that the sensitive data arrives.

Figure 3 gives an example of how *Permission Use Analysis* works. This piece of code monitors the location change event to send the latest location to a remote party via SMS. At point *1*, it registers a *SmsLocationListener* to monitor the location change event by invoking the API *LocationManager.requestLocationUpdates*. *Permission Use Analysis* identifies this API invocation as a sensitive interaction because it triggers a `ACCESS_FINE_LOCATION` permission check. Since, *LocationManager.requestLocationUpdates* API use asynchronous resource delivery mechanism to propagate location resource, *Permission Use Analysis* further recognizes its registered callback (point *2*) as a resource delivery point. When the location changes, point *2* is invoked to acquire the latest location. At point *3*, the acquired location is transformed into a *String* which is sent to the number "12345" via SMS. Similarly, *Permission Use Analysis* would identify it as a *E-PUP* of `SEND_SMS` permission. Meanwhile, *Permission Use Analysis* would also treat this point as an *I-PUP* of `ACCESS_FINE_LOCATION` permission for utilizing the acquired location resource.

According to the two phases in *Permission Use Analysis*, it is implemented by two main components: *E-PUP Identifier*, which identifies all *E-PUP*s with accurate permission information (details described in Section III-A); and *I-PUP*

Tracker, which keeps tracking of the resources requested at each *E-PUP* to trace all *I-PUP*s (details described in Section III-B).

A. E-PUP Identifier

During the execution, applications may request system resources that are protected by some permissions. *E-PUP*s represent such behaviors in the application. The key feature of an *E-PUP* is that it's a callsite that invokes an Android API, and a permission check occurs during the execution of this API. To reconstruct effective permission use behaviors, the *E-PUP Identifier* should meet two properties.

- **Completeness.** It should completely identify all the call-sites that invoke privileged Android APIs.
- **Accuracy.** It should catch accurate information about the permission checked by Android during the execution of an API; otherwise the correctness and preciseness of the reconstructed behaviors cannot be guaranteed.

Existing work [24] and [25] has built privileged API lists with required permissions. It seems that our *E-PUP Identifier* could leverage such API-permission lists to identify *E-PUP*s by intercepting all APIs during the execution, and then looking up the permissions that would be checked in an API-permission list by matching API signatures. Unfortunately, existing API-permission lists are either incomplete [24] or inaccurate [25]. Stowaway [24] uses Java reflection to execute Android APIs and monitors what permissions are checked by the system. To create appropriate arguments for each API, Stowaway uses API fuzzing to automatically generate test cases. Although Stowaway's API-permission list is accurate, it is quite incomplete due to the fuzzer's inability to generate complete inputs for all Android APIs. To achieve a good coverage, PScout [25] adopts static analysis to extract API-permission lists from Android source code. Although PScout's API-permission list is relatively complete, it is not accurate enough, because an Android API could use different permissions at runtime according to its arguments, which is also acknowledged by its authors [25]. To implement a *both* complete and accurate *E-PUP Identifier*, we need to design a new technique, as described below.

1) E-PUP Identification Strategy: Based on our definition of *E-PUP*, we propose a straightforward identification strategy. First, our technique identifies the *application-system interface*, which is a code boundary between application code and system code. Based on the *application-system interface*, *E-PUP Identifier* intercepts all calls to Android APIs. Then, by monitoring permission check events in Android's permission enforcement system during the execution of an API and propagating the exact permission check information to the application side, *E-PUP Identifier completely* identifies all the *E-PUP*s with *accurate* permission use information. Since the permission check information is sniffed from the permission enforcement system, this strategy also works when an Android API is invoked through Java reflection or Java Native Interface.

The *application-system interface* is recognized at every function call site by checking whether the caller is application code and the callee is system code. As Android apps

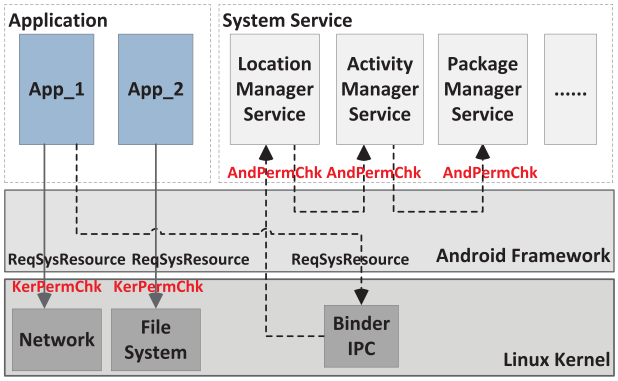


Fig. 4. Two kinds of permission checks in Android's permission enforcement system.

are mostly developed in the Java language and run on the Dalvik virtual machine, we instrument Dalvik to monitor all function calls. The algorithm to perform code origin checks should be very efficient, otherwise a huge performance penalty would be introduced. Fortunately, we find an efficient way to differentiate application code from system code by checking their class loader, because system code is loaded by a distinct class loader in Dalvik to ensure the VM integrity.

2) Acquire Permission Check Information: The complete identification of permission checks is the key to identify *E-PUP*s. With the permission check information, it's easy to judge whether an application-system interface is an *E-PUP* or a normal call site.

Android's permission system is enforced by two modules: Android system services and Linux kernel. According to the different permission enforcing techniques, we differentiate two kinds of permission checks in Android's permission enforcement system: *Android Permission Check (AndPermChk)* which is performed by Android system services to protect framework layer resources, and *Kernel Permission Check (KerPermChk)* which is enforced by Linux kernel to protect file system and network. For the two types of permission checks, the permission check information is propagated differently:

a) Propagate android permission check event: As Figure 4 shows, *AndPermChk* is performed in a separate Android process. The application side has no idea about what permission is checked by what system service. It is difficult to automatically propagate the permission check information from a separate service process to the application. Since Android apps employ Binder to invoke remote interfaces of a service process and the result is also returned via Binder, we choose to extend the Binder driver and its communication protocol to propagate the permission check information during the IPC procedure. As all *AndPermChk*s are finally handled by *ActivityManagerService*, we instrument its permission check logic to convey the permission check information to the Binder driver. With the extended Binder driver, this permission check information can be propagated back to the application side.

b) Propagate kernel permission check event: With a unique GID assigned to every kernel-enforced permission, *KerPermChk* is enforced by the GID isolation mechanism.

We instrument the GID isolation logic to record the checked GID into a kernel thread-local storage. The checked permission can be recognized by mapping the checked GID to the corresponding permission reversely. To acquire the permission check information from the kernel at the *application-system interface*, two system calls are added to access and clear the checked GID in the kernel thread-local storage. There may be some concerns that whether attackers would escape/attack the permission use identification by actively utilizing the two system calls. Actually, it would not. Since these two system calls are invoked just before or after the Android API, attackers could not tamper the execution flow during the API execution to invoke the two system calls.

Thus, with permission check information propagated to the application side, *E-PUP Identifier* could identify all *E-PUP*s with accurate permission use information.

B. I-PUP Tracker

While *E-PUP*s represent the behaviors of how an application use permissions to request sensitive resources, *I-PUP*s capture the internal behaviors of how the application manipulates these protected resources. To track the resources use points inside an app, *I-PUP Tracker* first needs to recognize the delivery point for each requested resource in the application.

1) Recognize Resource Delivery Point: Android's programming model complicates the identification of resource delivery points in the application. Callbacks are heavily used in Android to monitor privileged system events, such as location change events and phone state change events. There are three types of callbacks in Android that can be registered to deliver system resources: *BroadcastReceiver*, *PendingIntent*, and *Listener*. *BroadcastReceiver* is one of the four types of components defined in the Android application model, as described in Section II. *PendingIntent* [26] is a special Intent that can be sent back from a separate process on behalf of its creator. According to the ways of instantiating, a *PendingIntent* can be sent to an Activity, a Service or a BroadcastReceiver. *Listener* is a specialized class to handle callbacks that can be triggered remotely.

For most cases, *BroadcastReceivers* are declared in the app's manifest file and registered to the system when the app is installed. Android also provides APIs to register *BroadcastReceivers* at runtime. *PendingIntents* and *Listeners* are registered via specific Android APIs. Since callbacks are used by a small number of Android APIs, we choose to recognize the resource delivery point by monitoring those APIs that may register callbacks.

Although PScout's privileged API list [25] is not accurate enough for *E-PUP Identifier*, it provides a complete list for picking out APIs that register callbacks. However, there are more than 10,000 distinct APIs in PScout's API list for every Android version, so it is hard to manually check every API. Thus, we use an automatic method to filter out most APIs that definitely cannot register callbacks, and manually check a small number of remaining APIs.

Since only one specific API can register *BroadcastReceivers* at runtime, our automatic filtering method mainly selects APIs that register *PendingIntents* or *Listeners*. Our selection strategy is to find all potential APIs whose arguments may contain a *PendingIntent* or a *Listener*. We observe that *Listeners* can be invoked from a separate/remote process, so they are Binder objects. Firstly, our selection algorithm finds all the subclasses that extend *android.os.Binder*. Secondly, as an API may declare an interface as the argument type, our algorithm further collects a list for the interfaces that each Binder subclass implements. At last, our filtering method looks up PScout's API list to select those APIs with an argument type contained in the subclass list or the interface list.

TABLE II
CALLBACK APIs SELECTED IN POPULAR ANDROID VERSIONS

	Android 2.3	**Android 4.0**	**Android 4.1**
Total APIs	10906	15013	13009
Selected Callback APIs	286	319	338
New Callback APIs	–	65(33)	42(20)

Table II lists the callback APIs that are selected by the filtering method in several Android versions. For Android 2.3, our filtering method finds 232 APIs that may register *Listeners*. *PendingIntent* is easy to handle, because it is defined as a final class in Android. After a search on PScout's API list, our method finds 58 APIs whose arguments contain a *PendingIntent*. Then we manually verify the total 286 APIs (4 APIs register both *PendingIntents* and *Listeners*), and eventually we confirm 89 APIs register *PendingIntents* or *Listeners* to acquire protected system resources. For Android 4.0, there are 319 callback APIs selected. Compared with version 2.3, there are 65 newer callback APIs. For these new APIs, we need manually verify them to find whether they would register *PendingIntents* or *Listeners* for resource delivery. Actually, not all new APIs need to be manually verified, because many of them are just variants of old APIs in newer versions. Indeed, we just need to check 33 APIs for Android 4.0, and additional 20 APIs for Android 4.1. In this procedure, our automatic API filtering method greatly reduces the manual efforts and requires only small efforts to keep pace with the Android version.

For our selected APIs that register callbacks, the resource delivery point is the registered callback. While for other APIs, the *E-PUP* is also the resource delivery point. Since *BroadcastReceiver* can be registered by the manifest file, we parse the manifest file of each analyzed app to collect declared *BroadcastReceivers* and mark their *onReceive()* functions as the resource delivery points. After the resource delivery points are recognized, the *I-PUP*s can be tracked by following the resource usage inside the app.

2) Permission-Based Taint Analysis: After the resource is delivered to the application, it can be used in different ways with application-specific logic that makes the identification of *I-PUP*s quite difficult. To solve this problem, dynamic taint tracking can be used to capture the resource usage inside the application. However, traditional taint analysis relied on manually specified taint sources and annotated tags [1], thus it cannot be applied directly. The key challenge is to automatically taint related data for each delivered resource

with permission information. We propose permission-based taint analysis to address this problem. It works in the following steps.

a) Tag allocation: A taint tag is allocated at each *E-PUP* to mark the requested resource with corresponding permission check information. The taint tag is represented as a 32-bit integer. Each bit of the tag corresponds to a unique *E-PUP*. Our tag allocation is context-sensitive, which means the same tag will be assigned to *E-PUP*s with the same calling context. The reason for this strategy is to prevent the explosion of tag bits while different *E-PUP*s are still distinguishable.

b) Automatic data tainting: After a taint bit is allocated for an *E-PUP*, the corresponding acquired system resource needs to be automatically tainted with the tag. The automatic data tainting occurs at the resource delivery point for each *E-PUP*. For APIs that register callbacks, a wrapper is added around each registered callback to taint the delivered protected data according to the concrete type of the callback so that the related data gets tainted only when the callback is triggered. For other APIs, two kinds of data are automatically tainted according to the signature of the API: 1) The return value of the API at each *E-PUP* should be tainted with the corresponding tag. 2) As Java is an object-oriented language, the state of an object may be modified by instance methods. For instance APIs, we also taint the invoked object with the tag allocated at the *E-PUP*.

c) Identify I-PUPs: Dynamic taint tracking is employed to follow the propagation of tainted resource data. *I-PUP* is identified by recognizing the use point of tainted data. The granularity of the identification is quite important to the quality and efficiency of the *I-PUP Tracker*. It could be performed at the instruction-level, but a single instruction is too fine-grained to depict a meaningful action. Thus, we choose to identify *I-PUP* at the function-level. We intercept all function invocations in the Dalvik virtual machine and compute a taint tag for each function. The tag for a function is calculated by a bitwise OR operation on the taint tags of its parameter values. If the tag is non-zero, the function is an *I-PUP* for the permission represented by the tag.

After identifying resource delivery points and performing the permission-based taint analysis, *I-PUP Tracker* could trace all the use points of resources with accurate permission information.

IV. Auxiliary Components

A. Application Driver

Unlike traditional applications, there is no single entry point for an Android app. It brings problems to automatically executing Android apps. Considering the programming model of Android apps, our *Application Driver* adopts a component-based testing strategy. By parsing the manifest file of each Android app, it automatically extracts *Activities* and *Services* from the application and runs each component in the sandbox for a while (the time depends on the concrete hardware platform). Additionally, for each *Activity*, *Monkey* [27] is used to inject random UI events to exercise the user interface.

Furthermore, some behaviors of Android apps are triggered by events. Our *Application Driver* also injects fake events

```
1 private void _requestLocationUpdates(String provider,
2    Criteria criteria, long minTime, float minDistance,
3   boolean singleShot, LocationListener lis, Looper loop) {
4    ...//ommitted
5   if(VetDroid.isThisAppMonitored()){
6      VetLocationListener vetLis = new VetLocationListener(lis);
7      ListenerTransport trans = new ListenerTransport(vetLis, loop);
8      mService.requestLocationUpdates(provider, criteria, minTime,
9                  minDistance, singleShot, trans);
10     mService.fakeLocationChangeEvent();
11  }
12  ...//ommitted
13 }
```

Fig. 5. An example of injecting location change event.

(such as the arrival of new SMS, location change) during the execution of each component. As described in Section III-B, we extract a list of privileged APIs which are used to register callbacks. Based on this API list, we also instrument each API to notify the *Application Driver* module to inject related events when certain callbacks are registered through the API. Figure 5 shows an example about automatically injecting fake location change events when the corresponding callbacks are registered through the *LocationManager._requestLocationUpdates()* API. At Line 8, the location change monitoring request is delegated to *LocationManagerService* via the Binder protocol. Line 10 is our instrumentations logic which notifies *LocationManagerService* immediately to fake a location change event. Thus, the registered callback could be triggered for execution to enlarge the analysis scope. Compared with existing event injection technique, such as [28], our technique could perform *in-time* event injection at the most appropriate time, for not only callbacks listed in the Manifest but also API callbacks that invoked at runtime. With the runtime injected events, *Permission Use Analysis* module could reconstruct more permission use behaviors from the application.

Though many techniques are introduced to drive the application execution, it is worth noting that our *Application Driver* could not guarantee a complete coverage over all possible behaviors. In fact, this is generally a difficult problem for all dynamic analysis work. This paper tries to design a better behavior approximation for analyzing Android apps, and leaves the coverage problem as our future work.

B. Behavior Profiler

During the execution of the application in *VetDroid* sandbox, *Log Tracer* collects the behaviors reported by *Permission Use Analysis* module with runtime information to a log file. *Behavior Profiler* analyzes the log file offline to automatically generate permission use graphs for further analysis.

Behavior Profiler first identifies all the *E-PUP*s from the log file. For each *E-PUP*, *Behavior Profiler* further collects all *I-PUP*s for the requested permission by tracking the same tag bit. By connecting these permission use points according to the execution orders, *Behavior Profiler* could draw a permission use graph for each permission.

As Android adopts a fine-grained permission model [16] to protect system resources, our insight is that applications

usually need to use multiple permissions together to accomplish a meaningful behavior. Based on this observation, *Behavior Profiler* searches all the permission use graphs to connect those graphs with an overlapped node (which uses at least two permissions) to form a new permission use graph. The permission use graph with multiple permissions captures interesting behaviors for analysis, as will be demonstrated later in the evaluation. *Behavior Profiler* automatically discards permission use graphs that use only a single (less interesting) permission with the exception of those graphs using a high-risk permission such as `SEND_SMS`, `CALL_PHONE`. The profiled permission use graphs capture the behaviors of using permissions inside an application, especially when multiple permissions are intertwined. With such permission use graphs, experts could inspect the internal logic of Android apps to analyze suspicious behaviors, verify programming logic, etc.

V. Prototype & Evaluation

We implement prototypes of *VetDroid* based on Android version 2.3 and version 4.1. Currently, *VetDroid* supports running on Samsung Nexus S phones, Samsung Galaxy Nexus phones and emulators. Note that our techniques are not limited to any specific version. Most of our enhancements to Android lie in the Dalvik virtual machine and the Linux Kernel, whose architectures are quite stable. The prototype can be easily ported to a higher version. For example, we spent only 4 hours in updating *VetDroid* from version 2.3 to version 4.1.

A. Implementation Summary

E-PUP Identifier instruments the Dalvik virtual machine to intercept all API invocations, and enhances the Linux kernel as well as the Binder driver to acquire accurate permission use information at the application side. *I-PUP Tracker* modifies the Android framework to monitor registrations and invocations of application callbacks, and extends the taint tracking logic in TaintDroid [1] to implement the permission-based taint analysis (as described before). The *Application Driver* and *Behavior Profiler* are implemented in Python. In all, *VetDroid* modifies and enhances several main components in Android including the Linux kernel, the Binder driver, the Dalvik virtual machine, to implement a systematic permission use analysis framework.

We evaluate *VetDroid* from three aspects. We first apply *VetDroid* to real-world Android malware and analyze their internal malicious behaviors with permission use graphs. Next, we report our findings on vetting more than one thousand top free apps in Google Play with *VetDroid*. Finally, we measure the runtime overhead of *VetDroid* and the efficiency of tag allocation used in the taint analysis.

B. Real-World Malware Study

We have used *VetDroid* to analyze 600 Android malware samples that we have collected from Malware Genome Project [29]. To efficiently construct permission use behaviors, *Application Driver* runs these samples in 10 emulators and each component is executed for 120 seconds. Our hardware platform is an AMD server with 4*4 cores (2GHz) and 16GB memory. In all, 5,990 components are executed, which last totally about 22 hours (i.e., 2.2 minutes per sample). The reconstructed behaviors are automatically classified by their *E-PUP*s and further manually confirmed and categorized.

TABLE III
Example Behaviors Analyzed by VetDroid

Behaviors	#	Malware Families
Steal SMS	46	BaseBridge, SMSReplicator, Zitmo, Gone60
Steal Phone Number	38	ADRD, YZHC, GoldDream, Pjapps, GGTracker GingerMaster, DroidDream, DroidKungFu[1-4]
Steal Contact	8	Zitmo, Gone60, Walkinwat
Track Loc.	9	TapSnake, DroidDream, DroidKungFu1 Bgserv, DroidKungFu2, DroidKungFu4
Send SMS	43	Pjapps, Zsone, Walkinwat, RogueSPPush GGTracker, FakePlayer, SMSReplicator
Block SMS	22	Zitmo, RogueSPPush, GGTracker, Zsone

Table III lists six example categories of interesting malicious behaviors [29] captured by *VetDroid*. We can find that these malware either steals users' sensitive data or incurs financial charge. We also compare the analysis results with those reported by Malware Genome Project [29]. Unfortunately, the Command and Control (C&C) servers [30] used by some samples were not available during the analysis and some malicious behaviors are only triggered under certain contexts, so some behaviors reported in [29] were not observed. In all, *VetDroid* successfully analyzed 21 malware families and more importantly reconstructed their detailed behaviors, demonstrating its effectiveness in aiding malware analysis. More interestingly, *VetDroid* captured some previously unreported behaviors in dissected malware samples. For example, we found 38 BaseBridge samples exhibit *SMS Stealing* behavior and 1 Zitmo sample has *SMS Blocking* behavior, which have not been reported by Malware Genome Project yet.[1]

In the following, we present two interesting case studies of using *VetDroid*to analyze *GGTracker* and *SMSReplicator* samples. The permission use graphs capture the complete execution flow related to the malicious behaviors.[2] The nodes with filled colors represent *E-PUP*s, while other nodes represent *I-PUP*s. The edges in the graph depict the flow among permission use points.

1) Analysis of GGTracker: GGTracker is known for its intent to automatically sign up infected users to premium services. Due to the second-confirmation policy required in some countries, *GGTracker* needs to stealthily reply to an acknowledge SMS message sent from the service provider to sign up a premium-rate service. This behavior is critical to understand the internal logic of this malware.

We observe two kinds of behaviors in *GGTracker* with *VetDroid*. The first is the *SMS blocking* behavior. Figure 6 presents this behavior. We find *t4t.power.management.activity.SmsReceiver* is used to intercept any new SMS message. Then *getOriginatingAddress* is invoked to get

[1]Some of these behaviors have been mentioned in other places, see [31] has mentioned the *SMS Blocking* behavior in Zitmo samples.

[2]In this paper we only present partial permission use graphs.

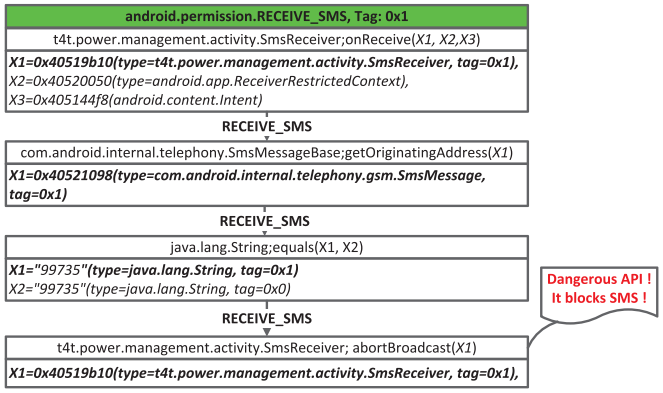


Fig. 6. *SMS blocking* behavior in *GGTracker*.

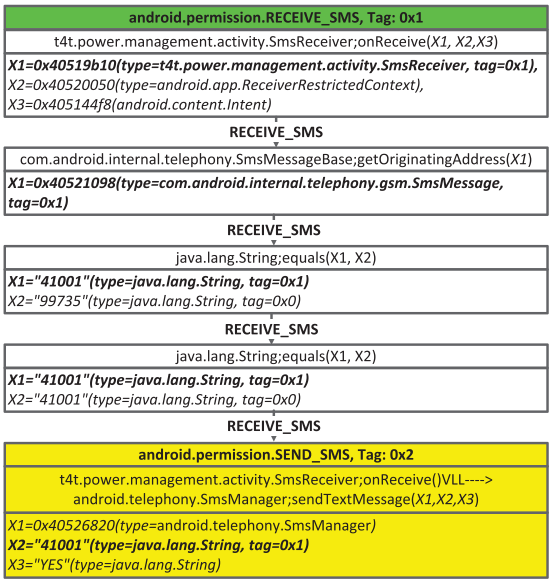


Fig. 7. *SMS Auto Reply* behavior in *GGTracker*.

the sender's number of the message. The permission use graph clearly shows that the malware would check whether the sender's number is "99735". By manually faking a SMS from "99735", we find this malware invokes *abortBroadcast()*. This API would suppress the broadcasting of the event about the arrival of a new SMS. Since *GGTracker* registers its BroadcastReceiver with the highest priority, this SMS is hidden from the user. Similarly, by checking the constraints on the sender's number from the graph, we can direct the *Application Driver* to inject faked SMS from other numbers (this can be easily implemented with an emulator [32]) to cover more interested behaviors. At last, we confirm *GGTracker* also blocks SMS from "46621", "96512", "33335", "36397", etc.

Besides, we also observe *SMS Auto Reply* behavior during iteratively changing the sender's number of the faked SMS. From Figure 7, we find when the malware intercepts a SMS from "41001", it automatically replies an SMS to "41001" with the content "YES" using the *sendTextMessage* API. The *SMS Auto Reply* behavior is critical in this kind of malware that stealthily signs up infected users to premium services. With *VetDroid*, this behavior is clearly revealed, enabling the detection and prevention of such attacks.

```
1  getpid() = 593
2  gettid() = 593
3  clock_gettime(CLOCK_MONOTONIC, {3234, 909151836}) = 0
4  ioctl("/dev/binder", BT819_FIFO_RESET_HIGH, ***) = 0
5  ioctl("/dev/binder", BT819_FIFO_RESET_HIGH, ***) = 0
6  clock_gettime(CLOCK_MONOTONIC, {3234, 924147234}) = 0
7  getpid() = 593
8  gettid() = 593
9  clock_gettime(CLOCK_MONOTONIC, {3234, 980469604}) = 0
10 ioctl("/dev/binder", BT819_FIFO_RESET_HIGH, ***) = 0
11 ioctl("/dev/binder", BT819_FIFO_RESET_HIGH, ***) = 0
12 clock_gettime(CLOCK_MONOTONIC, {3234, 993068739}) = 0
13 writev("/dev/log/radio", ***, 3) = 37
14 stat64("***/shared_prefs/phone.xml", 0xbee229e0) = -1
15 access("***/shared_prefs/phone.xml.bak", F_OK) = -1
16 access("***/shared_prefs/phone.xml", F_OK) = -1
17 stat64("***/shared_prefs/phone.xml", 0xbee229e0) = -1
18 stat64("***/shared_prefs/carrier.xml", 0xbee229e0) = -1
19 access("***/shared_prefs/carrier.xml.bak", F_OK) = -1
20 access("***/shared_prefs/carrier.xml", F_OK) = -1
21 stat64("***/shared_prefs/carrier.xml", 0xbee229e0) = -1
22 stat64("***/shared_prefs/content.xml", 0xbec249e8) = 0
23 getpid() = 593
24 gettid() = 593
25 clock_gettime(CLOCK_MONOTONIC, {3236, 498961380}) = 0
26 ioctl("/dev/binder", BT819_FIFO_RESET_HIGH, ***) = 0
27 ioctl("/dev/binder", BT819_FIFO_RESET_HIGH, ***) = 0
28 clock_gettime(CLOCK_MONOTONIC, {3236, 511990618}) = 0
29 getpid() = 593
30 gettid() = 593
31 clock_gettime(CLOCK_MONOTONIC, {3236, 744645490}) = 0
32 ioctl("/dev/binder", BT819_FIFO_RESET_HIGH, ***) = 0
33 ioctl("/dev/binder", BT819_FIFO_RESET_HIGH, ***) = 0
```

Fig. 8. System call list for *SMS Auto Reply* behavior.

a) Comparison with strace: To have a brief comparison with syscall-based analysis, we use *strace* to collect system call trace during the execution of *SMS Auto Reply* behavior (as listed in Figure 8). From the total 33 system calls, it's hard to recognize them as *SMS Auto Reply* behavior due to the loss of Android-level semantics and context information, while *VetDroid* can clearly reconstruct such behavior with the analysis of permission use points and behaviors.

b) Comparison with CopperDroid: CopperDroid [28] recovers Android-level semantics by dissecting Binder IPC. From the behavior report generated by CopperDroid for GGTracker [33], we could find SMS send behavior. However, we could not easily tell whether this behavior is an immediate result caused by the arrival of a new SMS message or not, nor can we easily understand the purpose of this malware.

2) Analysis of SMSReplicator: SMSReplicator [34] is a spyware app targeting infected users' incoming short messages. This malware protects itself by hiding its icon. *SMSReplicator* not only leaks SMS messages, but also incurs additional financial charge. From the permission use graph (Figure 9), we find that all the incoming SMS messages are intercepted by this malware using a *BroadcastReceiver* (*com.dlp.SMSReplicatorSecret.SMSReceiver*). Furthermore, *SMSReplicator* queries the contacts to find the sender of the intercepted message. Finally, the name of the sender and the message body is concatenated to send to a number specified by the attacker via SMS.

It is relatively easy to recognize this behavior as *SMS Forwarding*.

Similarly, we also check the behavior analyzed by CopperDroid on SMSReplicator. From [35], we could find CopperDroid detects the SMS interception and SMS Send

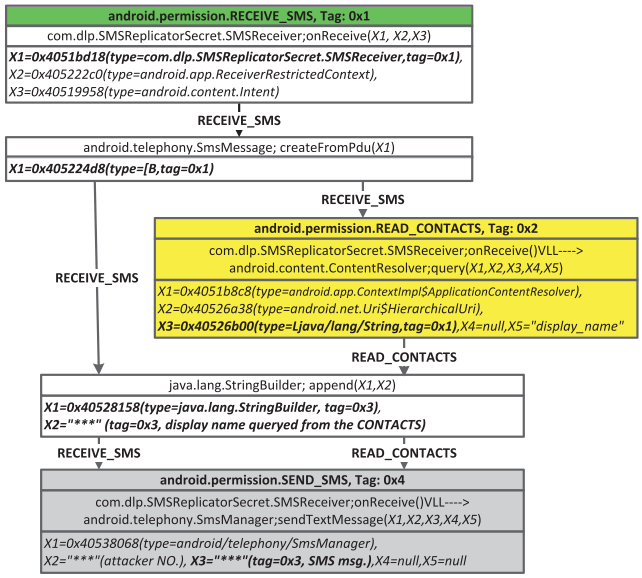


Fig. 9. *SMS Forwarding* behavior in *SMSReplicator*.

TABLE IV
INFORMATION LEAKAGE RESULTS

Leak Resource	TaintDroid	VetDroid
IMEI	135	135
Phone Number	7	7
Location	17	24
Network State	0	28

TABLE V
INFORMATION LEAK ANALYSIS RESULTS

Leak Resource	#	Leak Cause
Location	12	Inner-Active Ads Wetter Ads Flurry Ads Google Ads InMobi Ads Fortumo Payments
CellLocation	3	Vserv Ads Handmark
Phone Number	1	Mobile Public

behavior. However, with this report, we could not further identify fine-grained dynamics in malicious intents, i.e. whether it is *SMS Auto Reply* behavior or *SMS Forwarding* behavior. Meanwhile, with the reconstructed permission use behaviors which track the internal application logic, their divergent malicious intents are clearly differentiated by *VetDroid*.

C. Vetting Market Apps

Next, we use *VetDroid* to vet 1,249 top (benign) apps crawled from Google Play official store. These apps are top free apps crawled from 32 different categories such as games, education, entertainment, finance, social, sports, tools. We also use multiple emulators to parallelize the process of reconstructing permission use behaviors for these apps. There are several interesting findings.

Finding 1 (VetDroid Can Assist in Finding More Information Leaks Than TaintDroid): Based on the reconstructed permission use behaviors, we implement a simple permission-based filter that selects permission use graphs with at least one permission to read system resource and one permission to exfiltrate data to a remote party. The selected graphs are further classified with regard to *E-PUP*s. We manually check these classified behaviors and confirm four kinds of information leaks, as listed in Table IV.

We also use TaintDroid [1] to run these apps with the exact same inputs to the *Application Driver*. The results are also presented in Table IV. We can see that *VetDroid* detects 7 more location leaks than TaintDroid. After a further investigation on these cases, we find that the cell location (acquired through *TelephonyManager.getCellLocation()* API) is leaked in these cases while TaintDroid does not treat this kind of location as sensitive data. Since an app needs to use `ACCESS_COARSE_LOCATION` permission to get the cell location, *VetDroid* could automatically tracks the behaviors of leaking such kind of sensitive resource by following the permission usage. *VetDroid* also detects 28 cases that leak the device's network state to a remote party while TaintDroid's current implementation does not support detecting leaks of such sensitive resource.

It is worth noting that TaintDroid might be possibly improved to detect these leaks if we *proactively* and *manually* add ad-hoc logic to taint these sources. However, different from TaintDroid, *Permission Use Analysis* can *automatically* track such resources from the permission perspective. This experiment clearly demonstrates the necessity of adopting permissions to generally analyze sensitive behaviors.

Finding 2 (VetDroid Can Inspect the Fine-Grained Causes of Information Leakage): Permission Use Analysis captures the internal logic of permission usages inside an app, thus enables us to analyze the fine-grained procedure of information leakage. We manually analyze the permission use behaviors of several information leaks reported by *VetDroid* to investigate the contexts of reading and leaking sensitive information. In this experiment, we mainly focus on *Phone Number* and *Location* leakage cases because they are relatively interesting.

Based on the internal context of information leakage, we find that many such information leaks are actually not caused by the app itself. Table V shows our analysis results. From this table, we could find that 15 out of 24 location leaks are actually caused by mobiles Ads and payments. There is also one case that sends the phone number to a mobile promotion and publishing company (Mobile Public). Cell locations that are not tracked by TaintDroid are also used by Vserv and Handmark for better advertising.

With this experiment, we can see that *VetDroid* is capable of inspecting the fine-grained causes of sensitive information leakage by tracing internal permission use behaviors, while TaintDroid could only alert information leaks.

Finding 3 (VetDroid Can Help Detect Subtle Application Vulnerabilities): Since SMS service is unique and quite important for smartphones, we analyze 33 apps that request both `RECEIVE_SMS` and `SEND_SMS` permissions by running

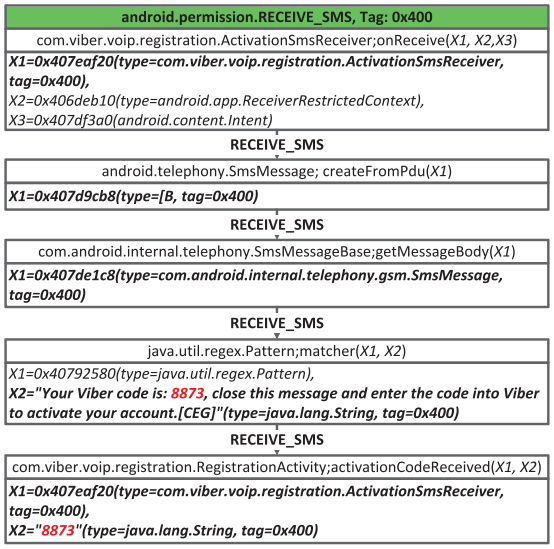


Fig. 10. *SMS Activation* behavior in Viber.

TABLE VI
RESULTS OF EXECUTION TIME AND MEMORY FOOTPRINT OVERHEAD ON CAFFEINEMARK BENCHMARK

	E-PUP	I-PUP	Log	ALL
Time	18.124%	10.385%	3.785%	32.294%
Mem.	0.100%	13.573%	0.637%	14.110%

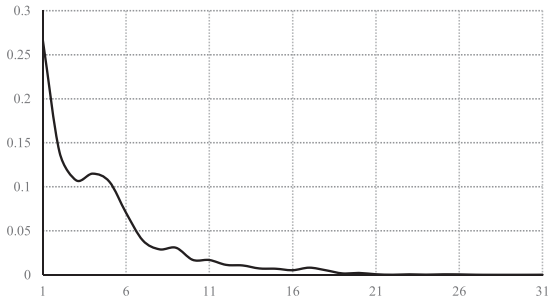


Fig. 11. Tag bit count distribution over the analyzed log files.

these apps in *VetDroid*. By carefully examining the permission use behaviors, we find that the Viber application is vulnerable to *Account Hijack attack*.

According to the website of Google Play, Viber is a free VoIP app that has been downloaded nearly 100 million times in recent 30 days worldwide. Viber provides users with free calls and messages to other Viber users. It also requests its user to bind his/her phone number which is used as his/her identity. When a call/message arrives, Viber will look up the sender's profile in the contact with the sender's phone number for a friendly notification.

To prevent a user from binding others' phone numbers, Viber server sends an activation SMS to the phone number. By verifying the activation code in the SMS, Viber can confirm whether the user owns the phone number or not. The activation phase is quite important for a popular communication app such as Viber. Otherwise, an attacker could bind a victim's phone number and send fake messages/calls to the victim's friends on behalf of the victim.

We use *VetDroid* to reconstruct the permission use behavior of the activation process. As Figure 10 shows, Viber intercepts incoming SMS messages in *ActivationSmsReceiver*, and extracts the activation code from the message body using a regular expression. Once an activation code is matched, the activation process is proceeded in the *RegistrationActivity.activationCodeReceived()* function.

By carefully examining the whole permission use behavior, we find that Viber does not check the origin of an activation SMS. Thus, an attacker could pass the activation by intercepting the activation SMS from the victim and sending it to the attacker's Viber client, causing the victim's account hijacked. It is not hard to steal an SMS from a victim, especially when the *Account Hijack attack* on the victim could lead to a reasonable profit. SMS stealing could be possibly implemented by malware such as *SMSReplicator* [34], *Zitmo* [36] or social engineering. To further confirm this vulnerability, we perform an experiment to hijack the Viber account of a volunteer in our group. By stealthily replacing an app in his smartphone into our repackaged version (which has the similar *SMS Blocking and Stealing* behavior as *Zitmo*), the activation SMS from Viber server is forwarded by our repackaged app to the attacker's device. After binding the volunteer's phone number to the attacker's device, free calls and messages are successfully initiated to his friends on behalf of his identity.

D. Performance Overhead Evaluation

Due to the inline instrumentation on Android, our analysis tool incurs some extra runtime overhead. We perform experiments on our Nexus S to measure the overhead from two aspects: execution speed and memory footprint. Table VI shows the results on CaffeineMark, a standard performance benchmark. Compared with the original Android system, *VetDroid* slows down the entire execution of the application by 32.294%, while increases the memory footprint by 14.110%. The main overhead of *I-PUP Tracker* is caused by our permission-based taint analysis which inherits the overhead of TaintDroid [1]. We believe this is a very reasonable and acceptable overhead for an offline analysis tool.

E. Tag Allocation in Permission-Based Taint Analysis

In the permission-based taint analysis, each bit in a 32-bit integer is used to represent a permission use point. Obviously, this raises concerns about whether 32 bits are enough for permission use point representation. We collect the tags allocated in 11,500 log files which are generated by *VetDroid* during the above analysis of malware and market apps, and present the results in Figure 11. This figure shows the percent of log files (vertical line) which need a number of tag bits specified by the horizontal line. From Figure 11, we could find that a 32-bit integer is sufficient for *E-PUP* identification. After a search over the 11,500 log files, we find there are totally 83,881 *E-PUP*s. By taking calling context

of *E-PUP* into account, 34,328 *E-PUP*s (which means more than 40 percent) do not need to be assigned with new tags. It means that the context-sensitive tag bit allocation strategy used in our technique greatly helps to prevent the explosion of tag bits.

VI. Discussion and Limitations

To effectively analyze Android apps, this paper proposes a new technique, called *Permission Use Analysis*. This technique adopts permission use perspective to identify sensitive interactions between the Android system and apps, and retrofits taint analysis for tracking sensitive application-specific behaviors in utilizing protected system resources. While taint tracking is an important part in our technique, *Permission Use Analysis* is more than a new kind of revised taint tracking. *Permission Use Analysis* solves the difficulties in *completely* identifying permission use points with *accurate* permission information and *automatically* attaching permission information with application-internal data before taint tracking.

By effectively abstracting sensitive behaviors in Android apps, *VetDroid* aims to be an effective offline analysis tool, instead of a malware detector or an Anti-Virus scanner. *VetDroid* could be very beneficial to security researchers and malware analysts with dissected fine-grained application behaviors. As our evaluation shows, *VetDroid* is not limited to analyze malicious apps, but also capable of analyzing benign apps. A key advantage of our approach is that it captures the application's sensitive behaviors with permission use graphs, which can significantly reduce irrelevant/uninteresting actions and let analysts focus on the critical behaviors when inspecting an app's internal logic. In practice, analysts can use *VetDroid* to automatically analyze a batch of apps and write simple scripts to select interested cases for further analysis (as demonstrated in our evaluation).

Compared with existing work, *Permission Use Analysis* provides a better approximation of sensitive behaviors inside an Android app. However, it bears limitations in the following aspects.

Our *I-PUP Tracker* is built upon TaintDroid, thus inheriting similar limitations of TaintDroid such as incapable of tracking implicit flows and native code. DTA++ [37] proposed targeted control-flow propagation to selectively track tainted control dependencies. However, this technique only applies to benign programs, and could not solve the problems of taint analysis depicted in [38]. Sarwar et al. [39] elaborate ways to evade TaintDroid such as control dependencies, side channels. It is still an open problem of handling implicit flows, so we leave it as future work. A possible way to solve the native code problem is to perform taint tracking at the binary level. However, it would incur a high overhead [22]. One possible way to solve this problem is to build a taint analysis system for mixed-language executions in which taint tracking of Java code is performed by Java virtual machine while native code is tracked by an emulator. Thus, the overhead of pure taint tracking at the binary level can be reduced. Furthermore, our *E-PUP Identifier* relies on the Android permission system for permission check identification. Thus our current implementation could not catch those behaviors that do not cause permission checks [40].

To enlarge the analysis scope, our *Application Driver* utilizes several key features of Android, such as component-based programming model, event triggers. However, our technique alone could not guarantee all possible behaviors are captured within the short time an app is executed. The *Application Driver* could be enhanced with an automatic input generation system such as AppsPlayground [41], AppInspector [42], Dynodroid [43], or guided analysis technique such as multi-path exploration [44], forced/informed execution [45], [46].

As a dynamic analysis technique, *VetDroid* has to observe the running of sensitive behaviors, which could be relatively slow compared with static solutions. To analyze apps at market-scale, Chakradeo et al. [47] proposed app triage to efficiently allocate malware analysis resources. *VetDroid* can be benefited by combining this technique into a practical market-scale application analysis solution.

VII. Related Work

We structure the related work from two aspects: traditional techniques for analyzing malware and permission-related analysis techniques.

A. Malware Analysis

Plenty of studies have focused on analyzing malware at the level of system call [12]. In [5], sequenced system calls with arguments were translated into actions that capture the sample's behaviors, such as changes to file system, modifying registries. Crowdroid [6] used system call vectors to represent the signature of malicious behaviors. The temporal pattern of system calls [7] was proposed to depict the application behavior for Symbian platform. Lanzi et al. [8] pointed out the limitation of reconstructing behaviors using linear system calls with a large-scale study. They reconstructed resource access behaviors by considering read/write system calls to identify malicious intents with the observation that most benign programs access their own files and registries. Dependency graphs of system calls were firstly proposed in [9] to represent behaviors. It captures the intrinsic application-system interactions and seems to be a good solution for behavior representation. In [9] and [10], researchers reconstructed dependencies among system calls by matching the types of their arguments and return values. Comparetti et al. [11] employed dynamic taint analysis to track the dependencies among system calls.

However, syscall-based techniques are not well-suited for the Android platform due to the inability of monitoring Android-specific behaviors. DroidScope [22] seems to notice these problems by seamlessly reconstructing the semantics from system calls and Java. However, it only refines existing work, leaving the root problems of Android's special permission mechanism and programming model untouched. A survey on current Android malware characteristics was presented in [29] and [48]. DroidRanger [49] and RiskRanker [50] were two Android malware detectors that relied on existing knowledge about malicious symptoms. Although they were reported to detect known and unknown malware samples, they

do not analyze the fine-grained internal behaviors of malware samples, which is the focus of *VetDroid*.

B. Permission Analysis

Felt et al. [16] studied the effectiveness of the time-of-use and install-time permission grant mechanism. This work was extended in [51] to provide guidelines for platform designers in determining the most appropriate permission-granting mechanism for a given permission. Permission-based security rules were used by Kirin [52] to design a lightweight certification framework that could mitigate malware at install time. Apex [53] and Saint [54] were two extensions to the Android's permission system by introducing runtime constraints on the granted permissions. MobileIFC [55] introduced context-aware policies for permission enforcement. In this permission model, permissions are granted depending on the device state, such as the GPS location or time of the day. This mechanism brings a new kind of flexibility and interesting security applications.

To help end users understand application behaviors at install time, AppProfiler [56] devised a two-step translation technique which maps API calls to high-level behavior profiles. While *VetDroid* also tries to provide better behavior understanding, it is a tool provided for different users (security analysts) and it uses a different new technique/perspective (permission use behavior) to precisely capture application-system interactions and sensitive behaviors inside an app. WHYPER [57] novelly leveraged Natural Language Processing (NLP) techniques to assess the risks of application by measuring whether the developers have explicitly explained the reasons for requiring permission-sensitive resources in its functional descriptions.

Barrera et al. [3] performed an empirical analysis on the expressiveness of Android's permission sets and discussed some potential improvements for Android's permission model. Felt et al. [24] proposed the first solution to systematically detect overprivileged permissions in Android apps and one-third of the applications in this study were found to be overprivileged. Probabilistic models of permission request patterns [58] or permission request sets [59] were also used to indicate the risk of new applications. To extract permission specifications for Android, Stowaway [24] used API fuzz testing while PScout [25] adopted static analysis on Android source code. However, these two permission specifications were limited in either completeness or preciseness, making them not well-suited for implementing *E-PUP Identifier*.

Permission re-delegation attack in Android was first introduced in [60] and [61]. Grace et al. [40] empirically evaluated the re-delegated permission leaks in pre-installed apps of stock Android smartphones. CHEX [62] and DroidChecker [63] were two tools that could detect such kind of capability leaks. Bugiel et al. [64] proposed system-centric and policy-driven runtime monitoring of communication channels between applications at both Android-level and kernel-level, which could prevent not only re-delegation attacks but also collusion attacks. Chen et al. [23] adopted static analysis to extract permission event graphs and examined the constraint conditions on events for each privileged API using model checking. However, it could not capture the internal logic of using permissions, especially when multiple permissions are intertwined.

CopperDroid [28] was an analysis tool to reconstruct Android-specific behaviors with syscall-level introspection. It might be more suited for large-scale automated analysis, while *VetDroid* to help a human analyst to understand much better internal behaviors of the analyzed malware. Our *VetDroid* differs from all existing work in that it provides the first systematic framework to analyze permission use behaviors.

VIII. Conclusion

This paper presents *VetDroid*, the first approach to perform accurate permission use analysis to vet undesirable behaviors. To construct permission use behaviors, this paper proposes a systematic framework that completely identifies explicit and implicit permission use points with accurate permission information. *VetDroid* is shown to be able to clearly reconstruct malicious behaviors of real-world apps to ease malware analysis. It can also assist in finding information leaks, analyzing fine-grained causes of information leaks, and detecting subtle vulnerabilities in regular apps. In all, *VetDroid* provides a better vehicle for analyzing and examining Android apps, which brings benefits to malware analysis/detection, vulnerability analysis, and other related fields.

References

[1] W. Enck *et al.*, "TaintDroid: An information flow tracking system for real-time privacy monitoring on smartphones," in *Proc. 9th USENIX Conf. OSDI*, pp. 1–6, 2010.

[2] *IDC: Android Market Share Reached 75% Worldwide in Q3 2012*. [Online]. Available: http://techcrunch.com/2012/11/02/idc-android-market-share-reached-75-wo%rldwide-in-q3-2012/, accessed May 7, 2013.

[3] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to Android," in *Proc. 17th ACM CCS*, 2010, pp. 73–84.

[4] *Mcafee Threats Report: Third Quarter 2012*. [Online]. Available: http:// www.mcafee.com / ca/ resources / reports / rp - quarterly - threat - q3-2012.pdf, accessed May 7, 2013.

[5] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proc. 5th Int. Conf. DIMVA*, Jul. 2008, pp. 108–125.

[6] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop SPSM*, 2011, pp. 15–26.

[7] A. Bose, X. Hu, K. G. Shin, and T. Park, "Behavioral detection of malware on mobile handsets," in *Proc. 6th Int. Conf. MobiSys*, 2008, pp. 225–238.

[8] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *Proc. 17th ACM Conf. CCS*, 2010, pp. 399–412.

[9] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. 6th Joint Meeting ESEC-FSE*, pp. 5–14, Sep. 2007.

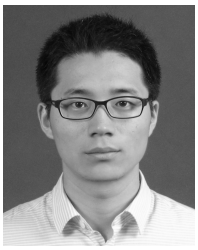
[10] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proc. IEEE Symp. SP*, May 2010, pp. 45–60.

[11] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying dormant functionality in malware programs," in *Proc. IEEE Symp. SP*, May 2010, pp. 61–76.

[12] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proc. ISSTA*, 2012, pp. 122–132.

[13] H.-G. Schmidt, K. Raddatz, A.-D. Schmidt, A. Camtepe, and S. Albayrak, "Google Android: A comprehensive introduction," DAI-Labor, Berlin, Germany, Tech. Rep. TUB-DAI 03/09-01, 2009.

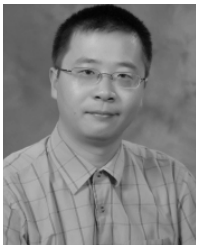
[14] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu, "Shadow attacks: Automatically evading system-call-behavior based malware detection," *J. Comput. Virol.*, vol. 8, nos. 1–2, pp. 1–13, May 2012.
[15] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *IEEE Security Privacy*, vol. 7, no. 1, pp. 50–57, Jan./Feb. 2009.
[16] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proc. 2nd USENIX Conf. WebApps*, 2011, p. 7.
[17] (2011). *Apple: iOS 4*. [Online]. Available: http://www.apple.com/iphone
[18] *Android Permissions*. [Online]. Available: http://developer.android.com/reference/android/Manifest.permission.html, accessed May 7, 2013.
[19] Y. Zhang *et al.*, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. ACM SIGSAC Conf. CCS*, 2013, pp. 611–622.
[20] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proc. ACM SIGSAC Conf. CCS*, 2013, pp. 1043–1054.
[21] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: Multi-layer profiling of Android applications," in *Proc. 18th Annu. Int. Conf. Mobicom*, 2012, pp. 137–148.
[22] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Security Symp.*, 2012, pp. 569–584.
[23] K. Z. Chen *et al.*, "Contextual policy enforcement in Android applications with permission event graphs," in *Proc. NDSS*, Feb. 2013.
[24] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. CCS*, 2011, pp. 627–638.
[25] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. ACM Conf. CCS*, 2012, pp. 217–228.
[26] *Pendingintent*. [Online]. Available: http://developer.android.com/reference/android/app/PendingIntent.html, accessed May 7, 2013.
[27] *Ui/Application Exerciser Monkey*. [Online]. Available: http://developer.android.com/tools/help/monkey.html, accessed May 7, 2013.
[28] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors," in *Proc. EuroSec*, Apr. 2013.
[29] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. SP*, May 2012, pp. 95–109.
[30] H. R. Zeidanloo and A. A. Manaf, "Botnet command and control mechanisms," in *Proc. 2nd ICCEE*, Dec. 2009, pp. 564–568.
[31] A. Apvrille and K. Yang, "Defeating mTANs for profit—Part one," *Virus Bull. Mag.*, vol. 7, no. 4, pp. 4–10, Apr. 2011.
[32] *SMS Emulation Using the Android Emulator*. [Online]. Available: http://developer.android.com/tools/devices/emulator.html#sms, accessed May 7, 2013.
[33] *CopperDroid Analysis Report for GGTracker*. [Online]. Available: http://copperdroid.isg.rhul.ac.uk/copperdroid/view.php?id=4514, acces sed May 7, 2013.
[34] *Android.SMSReplicator*. [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2010-110214%-1252-99, accessed May 7, 2013.
[35] *CopperDroid Analysis Report for SMSReplicator*. [Online]. Available: http://copperdroid.isg.rhul.ac.uk/copperdroid/view.php?id=5378, accessed May 7, 2013.
[36] *ZeuS-in-the-Mobile—Facts and Theories*. [Online]. Available: http://www.securelist.com/en/analysis/204792194/ZeuS_in_the_Mobile_Fact_and_Theories, accessed May 7, 2013.
[37] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. NDSS*, Feb. 2011.
[38] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Proc. 5th Int. Conf. DIMVA*, Jul. 2008, pp. 143–163.
[39] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices," in *Proc. 10th Int. Conf. SECRYPT*, 2013, pp. 461–467.
[40] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. NDSS*, 2012.
[41] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *Proc. 3rd ACM Conf. CODASPY*, 2013, pp. 209–220.
[42] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, "Vision: Automated security validation of mobile apps at app markets," in *Proc. 2nd Int. Workshop MCS*, 2011, pp. 21–26.
[43] A. MacHiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," Program Anal. Group, Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep. GIT-CERCS-12-09, 2012.
[44] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proc. IEEE Symp. SP*, May 2007, pp. 231–245.
[45] J. Wilhelm and T.-C. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *Proc. 10th Int. Conf. RAID*, 2007, pp. 219–235.
[46] Z. Xu, L. Chen, G. Gu, and C. Kruegel, "PeerPress: Utilizing enemies' P2P strength against them," in *Proc. ACM Conf. CCS*, 2012, pp. 581–592.
[47] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "MAST: Triage for market-scale mobile malware analysis," in *Proc. 6th ACM Conf. WiSec*, 2013, pp. 13–24.
[48] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proc. 1st ACM Workshop SPSM*, 2011, pp. 3–14.
[49] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. NDSS*, 2012.
[50] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. 10th Int. Conf. MobiSys*, 2012, pp. 281–294.
[51] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner, "How to ask for permission," in *Proc. HotSec*, 2012.
[52] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. 16th ACM Conf. CCS*, 2009, pp. 235–245.
[53] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android permission model and enforcement with user-defined runtime constraints," in *Proc. 5th ASIACCS*, 2010, pp. 328–332.
[54] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in Android," in *Proc. ACSAC*, Dec. 2009, pp. 340–349.
[55] K. Singh, "Practical context-aware permission control for hybrid mobile applications," in *Proc. 16th Int. Symp. RAID*, Oct. 2013, pp. 307–327.
[56] S. Rosen, Z. Qian, and Z. M. Mao, "AppProfiler: A flexible method of exposing privacy-related behavior in Android applications to end users," in *Proc. 3rd ACM CODASPY*, 2013, pp. 221–232.
[57] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. USENIX Security Symp.*, pp. 527–542, Aug. 2013.
[58] M. Frank, B. Dong, A. P. Felt, and D. Song, "Mining permission request patterns from Android and Facebook applications," in *Proc. IEEE 12th ICDM*, Dec. 2012, pp. 870–875.
[59] H. Peng *et al.*, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. ACM CCS*, 2012, pp. 241–252.
[60] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. USENIX Security Symp.*, 2011.
[61] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proc. USENIX Security Symp.*, 2011.
[62] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. CCS*, 2012, pp. 229–240.
[63] P. P. F. Chan, L. C. K. Hui, and S. M. Yiu, "DroidChecker: Analyzing Android applications for capability leak," in *Proc. 5th ACM Conf. WISEC*, 2012, pp. 125–136.
[64] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proc. NDSS*, Feb. 2012.



Yuan Zhang received the Ph.D. degree from Fudan University, Shanghai, China, in 2014, and the B.Eng. degree from Nanjing University, Nanjing, China, in 2009. He will join Fudan University as an Assistant Professor in Fall 2014. His research interests include system security and compiler techniques.

Min Yang is currently an Associate Professor with Software School, Fudan University, Shanghai, China. He received the B.Sc. and Ph.D. degrees in computer science from Fudan University, in 2001 and 2006, respectively. His research interests are in system software and security.

Zhemin Yang holds a post-doctoral position with Software School, Fudan University, Shanghai, China. He received the B.Sc. and Ph.D. degrees in computer science from Fudan University in 2007 and 2012, respectively. His research interests include Java virtual machine, maintainability and scalability of parallel systems, and concurrent software debugging.



Guofei Gu is currently an Associate Professor with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA. He received the Ph.D. degree in computer science from the College of Computing, Georgia Institute of Technology, Atlanta, GA, USA. His research interests are in network and system security, social Web security, and cloud and software-defined networking (SDN/OpenFlow) security.

Peng Ning is a Professor with the Department of Computer Science, North Carolina State University, Raleigh, NC, USA. He is currently on leave at Samsung Mobile, Santa Clara, CA, USA, where he is leading the Samsung KNOX Research and Development Team. His research interests are primarily in mobile security, wireless security, and cloud computing security.

Binyu Zang received the Ph.D. degree in computer science from Fudan University, Shanghai, China, in 1999, where he is currently a Professor. His research interests are in compilers, computer architecture, and systems software.