Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis

Yuan Zhang[1] Min Yang[1] Bingquan Xu[1] Zhemin Yang[1]
Guofei Gu[2] Peng Ning[3] X. Sean Wang[1] Binyu Zang[1]

[1]School of Computer Science, Fudan University, China
{yuanxzhang, m_yang, xubingquan, yangzhemin, xywangCS, byzang}@fudan.edu.cn
[2]SUCCESS Lab, Texas A&M University, USA, guofei@cse.tamu.edu
[3]Department of Computer Science, North Carolina State University, USA, pning@ncsu.edu

Abstract

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 reconstructing sensitive behaviors in Android apps from a novel permission use perspective. *VetDroid* features a systematic framework 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 1,249 top free apps in Google Play. *VetDroid* can assist in finding more information leaks than TaintDroid [24], 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 identify subtle vulnerabilities in some (top free) applications otherwise hard to detect.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.1 [**Software Engineering**]: Requirements/Specifications

Keywords

Android security; permission use analysis; vetting undesirable behaviors; Android behavior representation

CCS'13, November 4–8, 2013, Berlin, Germany.

l http://dx.doi.org/10.1145/2508859.2516689.

1. INTRODUCTION

Smartphone platforms are becoming more and more popular these days [5]. To protect sensitive resources in the smartphones, permission-based isolation mechanism [13] 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 [6], 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.

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, e.g., [14, 16, 21, 22, 33, 40, 49]) 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 [17]. However, previous work is not readily applicable due to the following unique features of Android:

Android Framework Managed Resources. Android is an application framework on top of Linux kernel [51] 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 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.

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. The wide use of IPC also brings problems 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 [42].

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. This asynchronous resource access model via system delivery is quite different from the synchronous application-request access model. A key observation is that application registered callbacks are application code, so they could evade system call interception. As a result, traditional behavior reconstruction methods will lose such important application behaviors.

The above analysis indicates that a general method to reconstruct 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 to reconstruct sensitive behaviors for Android apps from a novel *permission use* perspective, i.e., how applications use permissions to interact with the Android system and sensitive resources. 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. Accordingly, we define two kinds of permission use points: *Explicit permission use points (E-PUP)* denote those callsites of Android APIs in applications that explicitly request the permission-sensitive resources; *Implicit permission use points (I-PUP)* denote the use points of the acquired sensitive resources that are requested with permissions. For example, assume an application requests both `ACCESS_FINE_LOCATION` and `INTERNET` permissions during the installation time. Its permission use behavior 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 construct permission use behaviors for Android apps. *VetDroid* features a systematic permission use analysis to identify a *complete* set of *E-PUP*s and *I-PUP*s with *accurate* permission use information during the runtime. Our proposed permission use analysis is composed of two components: *E-PUP Identifer* which intercepts all invocations to Android APIs and sniffs accurate permission check information from Android's permission enforcement system to identifies all the *E-PUP*s with accurate permission use information, and *I-PUP Tracker* which takes the asynchronous resource delivery model into account to recognize the exact delivery point in the application for each resource requested at a *E-PUP* and locates all the *I-PUP*s of these resources by 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 behavior 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 [24], 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 behavior 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. §2 introduces some background information about Android and defines the permission use behavior. §3 describes our overall behavior reconstruction approach. After that, we present our evaluation results in §4 and discuss possible limitations & further improvements in §5. Finally, we discuss related work in §6 and conclude our paper in §7.

2. PROBLEM STATEMENT

2.1 Android Background

Android is the most popular mobile operating system today. It is built on top of more than 100 open source projects including Linux kernel. 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). The system components are also isolated into distinct identities. With the help of the identity isolation mechanism in Linux, applications in Android are isolated from each other and from the system.

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 [26]. 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 [30] which is different from iOS [11]. In Android 4.2, there are 130 items of sensitive resources that are protected with permissions [1].

The Android application framework forces a component-based application model [26] to increase the code reusability. It does not have a main() function or any single entry point for execution. Instead, Android apps must be developed in terms of components. There are four types of components defined in Android's programming model: *Activity* component has a user interface and handles the interactions with user, *Service* component performs background processing, *ContentProvider* component stores and shares data such as a relational database, and *BroadcastReceiver* component handles messages from other components, including the system. The primary mechanism for component interactions is through an *Intent*, which is simply a message object encapsulating the information of interest to the component that receives the intent such as the action to be taken and the data to act on, and some meta data managed by Android system. A component can be protected by permissions and only those applications with granted permissions can interact with the privileged component.

2.2 Motivation

Existing work [14, 21, 33, 40, 49] 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 §1, 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.

TaintDroid [24] alerts information leaks inside an Android app via dynamic taint tracking. AppIntent [57] 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 [53] is a behavior profiling system for Android apps which is also not suitable for analyzing internal behavior logic. DroidScope [56] 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 [20], 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 [13], which we believe to be important to understand behaviors of these applications. Thus, in this paper we propose to reconstruct permission use behaviors as a new and complementary aspect in analyzing Android apps.

2.3 Definition of Permission Use Behavior

Permission use behaviors aim to capture apps' internal sensitive behaviors on utilizing system resources that are protected by some permissions. According to the lifecycle of utilizing system resources inside an app, we define different kinds of permission use points (PUP). First, an app needs to invoke some Android APIs to request system resources, which we call *resource request stage*. If the requested resources are protected by some permissions, Android's permission enforcement system will check whether this app has been granted the corresponding permissions at install time. Because permission checks explicitly occur during *resource request stage*, we denote the callsites in the app that invoke Android APIs to request protected system resources as *explicit permission use points (E-PUP)*.

After the *resource request stage*, system resources may be delivered to the app synchronously or asynchronously, depending on the API used to request resources. The *resource delivery point* is the starting point to learn the behaviors of utilizing sensitive resources inside an app, and thus is very important for reconstructing permission use behaviors.

Finally, when the requested resources have been delivered to the app, they may be processed by application-specific logic, which reflects the internal behaviors of utilizing sensitive resources. For example, the location resource may be used by an app to suggest the restaurants nearby, or may be used by a malicious entity to track the victim. Although the further processing of acquired resources in an app does not cause additional permission checks against the app, it is still important to track the further use of these resources. In this paper, these internal use points of the protected resources are denoted as *implicit permission use points (I-PUP)*. *I-PUP*s make the critical behaviors stand out from other irrelevant application-specific actions to ease the analysis of the app.

Permission Use Behavior. As described above, *E-PUP*s capture what and where permissions are used by the application, while *I-PUP*s capture how the application uses permissions to implement their specific logic. However, a single permission use point only represents a sensitive action performed by the application, and does not necessarily capture a meaningful behavior for analyzing applications. Based on the *E-PUP*s and *I-PUP*s, we now formally define permission use behaviors.

DEFINITION 1. *A Permission Use Behavior is a function call graph $G = (V, E, \alpha)$ over a set of permissions P where:*

- *the set of vertices $V = V_{E-PUP} \cup V_{I-PUP}$, and it consists of all E-PUPs and I-PUPs,*
- *the set of edges $E \sqsubseteq V \times V$, and each edge connects nodes that use the same permission,*
- *the labeling function $\alpha : V \rightarrow P$, and it associates each node with permission(s) it uses.*

With permission use behaviors, the interactions between applications and the Android system are effectively abstracted because it describes how applications request system resources and internally use the acquired system resources. However, the two kinds of permission use points are hard to identify due to some unique features of Android and application-specific logic. We thus design an analysis platform called *VetDroid* to automatically reconstruct permission use behaviors from Android apps.

3. VETDROID DESIGN

The overview of *VetDroid* design is shown in Figure 1. Sample applications are first loaded into *Application Driver*, which automatically executes the application in our sandbox (details described in §3.3). During the execution, *Permission Use Analysis* module identifies all the *E-PUP*s, *I-PUP*s 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 §3.4).

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

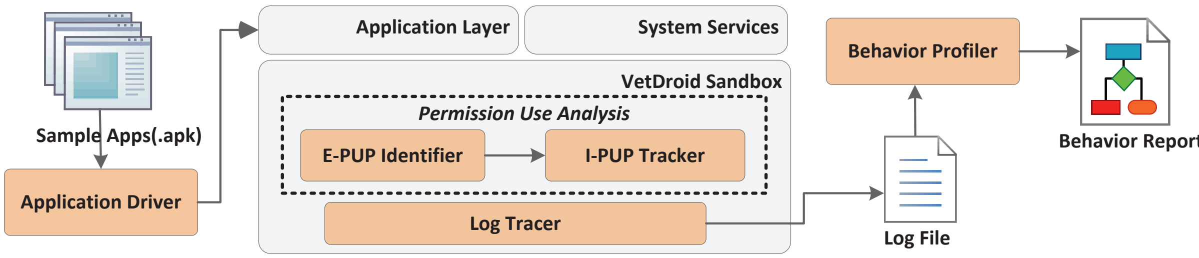

Figure 1: Overview of *VetDroid* to reconstruct permission use behaviors for Android apps.

and *precisely* track their relationships. To correctly capture the behaviors of using permissions inside an app, we analyze the execution flow of the application with regards to Android's special permission mechanism and programming model. Our systematic permission use analysis contains two main components: *E-PUP Identifier*, which identifies all *E-PUP*s with accurate permission information (details described in §3.1); and *I-PUP Tracker*, which keeps tracking of the resources requested at each *E-PUP* to trace all *I-PUP*s (details described in §3.2).

3.1 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 have two properties. First, it should completely identify all the callsites that invoke privileged Android APIs. Second, 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 [12,27] 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 [27] or inaccurate [12]. Stowaway [27] 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 [12] 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 [12]. To implement a *both* complete and accurate *E-PUP Identifier*, we need to design a new technique, as described below.

3.1.1 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* could intercept all calls to Android APIs. Then, by monitoring permission check events in Android's

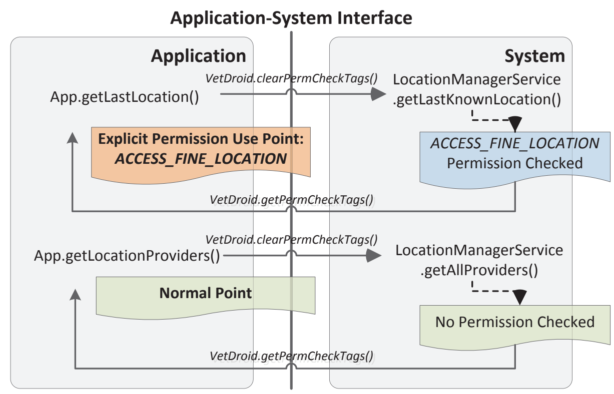

Figure 2: An example of identifying *E-PUP*s.

permission enforcement system during the execution of an API and propagating the exact permission check information to the application side, *E-PUP Identifier* could *completely* identify all the *E-PUP*s with *accurate* permission use information, including those invoked through Java reflection or Java Native Interface.

Figure 2 shows an example of identifying *E-PUP*s at the application side. In this example, *App.getLastLocation()* invokes *getLastKnownLocation()* API of *LocationManagerService* to get the last known location. Before invoking this API, *VetDroid* clears the permission check information in the thread-local storage using *VetDroid.clearPermCheckTags()*. During the execution of this API, Android's permission enforcement system performs a permission check on `ACCESS_FINE_LOCATION` permission. At last, after the execution of *getLastKnownLocation()* API, *VetDroid* invokes *VetDroid.getPermCheckTags()* to propagate the permission check information from the enforcement system to the application side. With the propagated permission check information, this callsite in *App.getLastLocation()* is identified as an *E-PUP* of `ACCESS_FINE_LOCATION` permission.

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 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.

3.1.2 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

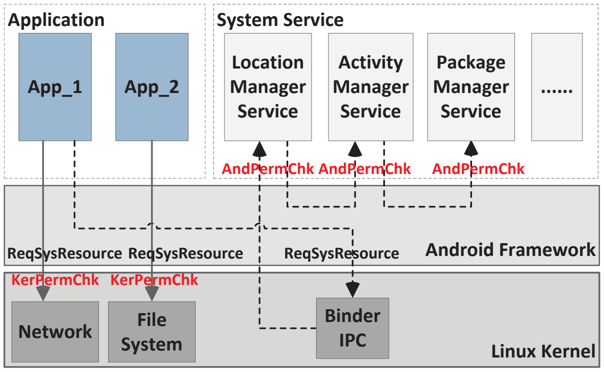

Figure 3: Two kinds of permission checks in Android's permission enforcement system.

to judge whether an application-system interface is an *E-PUP* or a normal call site (see *App.getLocationProviders()* in Figure 2).

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* and *Kernel permission check*. Figure 3 illustrates these two kinds of permission checks:

Android Permission Check (AndPermChk). When an app tries to access system resources that are protected by Android system services such as contacts and locations, *AndPermChk*s occurred. Figure 3 gives an example of *AndPermChk*. *App_1* tries to acquire the current location by invoking an interface of *LocationManagerService* via Binder. *LocationManagerService* first checks whether *App_1* has been granted `ACCESS_FINE_LOCATION` permission by invoking the general permission check interface of *ActivityManagerService*. The *AndPermChk* requests are finally redirected to *PackageManagerService* except the permission requests from the system itself are granted immediately. *PackageManagerService* handles the permission check request by looking up a table that records all the granted permissions for each application when it is installed. According to the permission check result, *LocationManagerService* judges whether to accept or deny the request from *App_1*.

Kernel Permission Check (KerPermChk). The permissions to protect file system and network are enforced by the Linux kernel. As Figure 3 shows, the accesses to these resources should pass *KerPermChk*s. In Android, a unique GID is assigned to each kernel-enforced permission. An app is checked to verify whether it has the corresponding GID before accessing the protected resource.

Our identification of permission checks is implemented in Android's permission enforcement system, while *E-PUP Identifier* needs to acquire permission check information at the application side to judge whether a callsite is an *E-PUP* and what permission is used by an *E-PUP*. For the two types of permission checks, the permission check information is propagated differently:

Propagate AndPermChk Information. As Figure 3 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.

Propagate KerPermChk Information. 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.

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

3.2 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.

3.2.1 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 §2. *PendingIntent* [7] 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 [12] 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. Our selection algorithm first finds all the subclasses that extend *android.os.Binder*. 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. 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. In this procedure, our automatic API filtering method greatly reduces the manual efforts.

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.

3.2.2 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, we use dynamic taint tracking to capture the resource usage inside the application. However, traditional taint analysis cannot be applied directly. The key challenge is to automatically taint related data for each delivered resource with permission information. Our permission-based taint analysis works in the following steps.

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.

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*.

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.

3.3 Application Driver

Unlike traditional applications, there is no single entry point for an Android app. It brings problems to automatically executing Android apps. Our *Application Driver* adopts a component-based testing strategy. 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, *Monkey* [9] is used to exercise the user interface for each *Activity*.

Furthermore, some behaviors of Android apps are triggered by events. Our *Application Driver* also injects fake events (such as the arrival of new SMS, location change) during the monitoring when certain callbacks are registered. With the runtime injected events, *Permission Use Analysis* module could reconstruct more permission use behaviors from the application.

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 (as discussed in §5).

3.4 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 [30] 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.

4. PROTOTYPE & EVALUATION

A prototype of *VetDroid* is implemented based on Gingerbread (Android 2.3).[1] This prototype currently supports running on

[1]Note that our techniques are not limited to this specific version.

Samsung Nexus S phones and emulators. The *Application Driver* and *Behavior Profiler* are implemented in Python. *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 [24] to implement the permission-based taint analysis (as described before). 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*.

4.1 Real-World Malware Study

We have used *VetDroid* to analyze 600 Android malware samples that we have collected from Malware Genome Project [59]. 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 1 lists six example categories of interesting malicious behaviors [59] 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 [59]. Unfortunately, the Command and Control (C&C) servers [58] 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 [59] 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. This further illustrates the advantages of our new analysis technique to help reveal undesirable behaviors.

Due to space limit, we can only present some interesting case studies analyzed by *VetDroid* with permission use graphs: *GGTracker*, *SMSReplicator*, *TapSnake*. 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.

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

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 1: Example behaviors analyzed by *VetDroid*

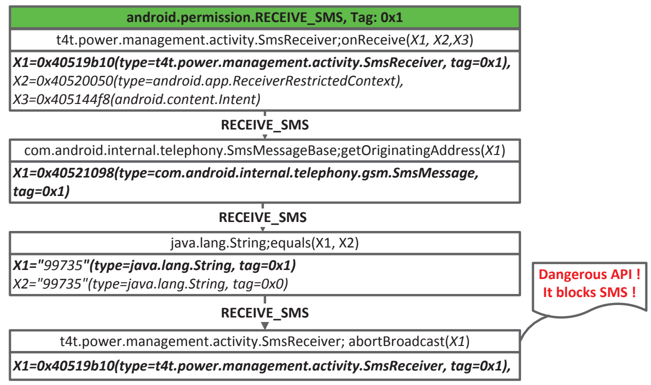

Figure 4: *SMS Blocking* behavior in *GGTracker*.

We observe two kinds of behaviors in *GGTracker* with *VetDroid*. Figure 4 shows the *SMS blocking* behavior. When a new SMS arrives, *t4t.power.management.activity.SmsReceiver* is triggered. Then *getOriginatingAddress* is invoked to get the sender's number of this message. The permission use graph clearly expresses the constraints on the sender's number in this malware. If this SMS is sent from "99735", this message is blocked by invoking *abortBroadcast()*. This function suppresses 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. 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 [8]) 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 by iteratively changing the sender's number of the faked SMS. From Figure 5, we could find that 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.

System Call Trace. 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 showed in Table 2. From the collected 33 system calls, it's hard to recognize them as *SMS Auto Reply* behavior due to the loss of fine-grained semantic and context information, while *VetDroid* can clearly reconstruct such behavior with the analysis of permission use points and behaviors.

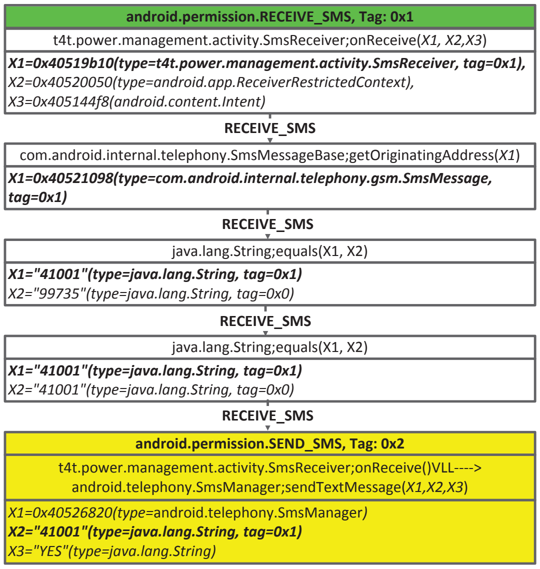

Figure 5: ***SMS Auto Reply*** **behavior in** ***GGTracker*****.**

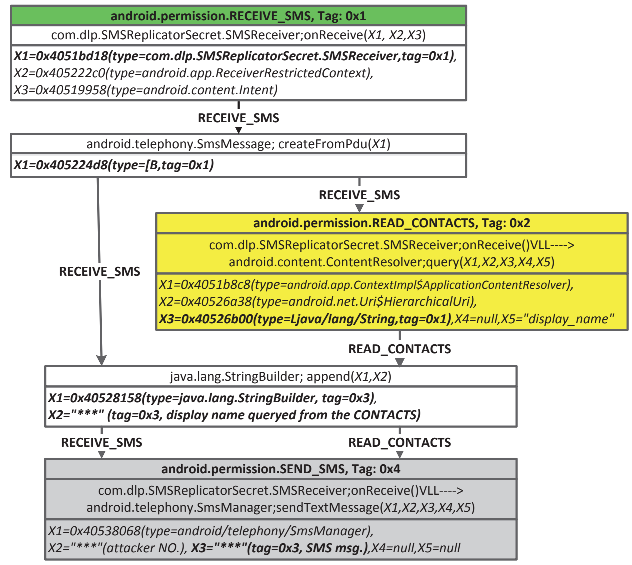

Figure 6: ***SMS Forwarding*** **behavior in** ***SMSReplicator*****.**

syscall	#	Comments
ioctl	8	Binder communication
stat64,access	9	app's resource file
getpid,gettid	8	process information
clock_gettime	7	time information
writev	1	log operation

Table 2: System call trace for ***SMS Auto Reply*** **behavior.**

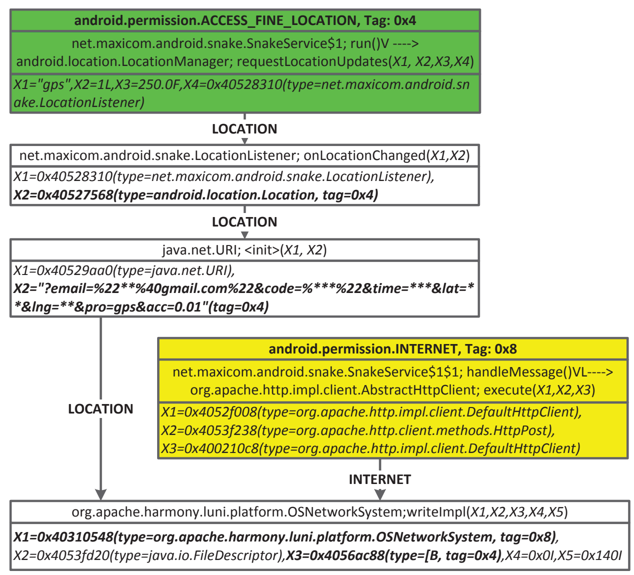

Figure 7: ***Location Track*** **behavior in** ***TapSnake*****.**

2) Analysis of SMSReplicator.

SMSReplicator [3] 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. As Figure 6 shows, all the incoming SMS messages are intercepted by this malware using a *BroadcastReceiver* (*com.dlp.SMSReplicatorSecret.SMSReceiver*). The SMS is instantiated using *createFromPdu()* function of *SmsMessage*. *SMSReplicator* further queries the contacts to find the sender of the intercepted message. The name of the sender and the message body is concatenated to send to a number specified by the attacker via SMS. This graph clearly shows the permission use points of three critical permissions (`RECEIVE_SMS`, `READ_CONTACTS`, `SEND_SMS`). It is relatively easy to recognize this behavior as *SMS Forwarding*. We can find that *SMS Auto Reply* behavior and *SMS Forwarding* behavior are similar in intercepting and sending SMS. However, with the reconstructed permission use behaviors which track the internal application logic (Figure 5 and Figure 6), their divergent malicious intents get clearly differentiated.

3) Analysis of TapSnake.

TapSnake [2] tracks the infected user by sending the latest location to a remote server. To hide its malicious intent, this malware disguises itself as the classic "snake" video game. During the installation, this malware asks users to grant `ACCESS_FINE_LOCATION` and `INTERNET` permissions. Considering these permissions are required by most legitimate advertising libraries [37], most users choose to grant these permissions without any idea about how these permissions will be used.

As showed in Figure 7, *TapSnake* first registers a callback (*net.maxicom.android.snake.LocationListener*) for the location change event. When location changes, the *onLocationChanged* function is invoked asynchronously by Android to deliver the latest location. *TapSnake* further performs some string operations on the location object to encode the location into a URL. The encoded URL is passed to the *execute()* function of *AbstractHttpClient*. The latest location that is encoded in the URL is eventually exfiltrated to the server *http://gpsdatapoints.appspot.com*.

4.2 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.

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

Table 3: Information leakage 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

Table 4: Information leak analysis results.

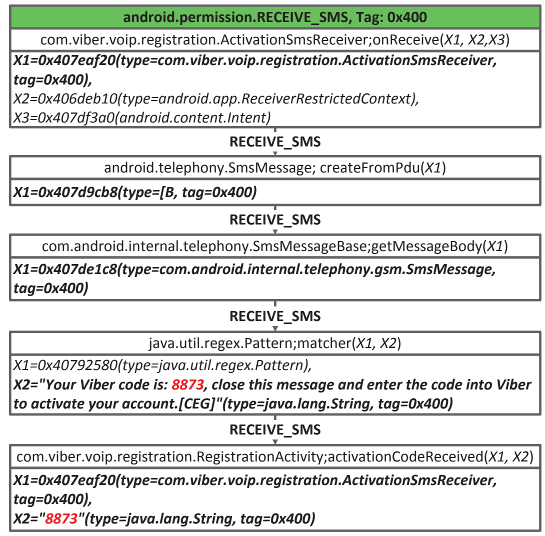

Figure 8: *SMS Activation* behavior in Viber.

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 3.

We also use TaintDroid [24] to run these apps with the exact same inputs to the *Application Driver*. The results are also presented in Table 3. 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 could be improved to detect these leaks if we *proactively* and *manually* add ad-hoc logic to taint these sources. However, different from TaintDroid, *VetDroid* can *automatically* track such resources as long as they are in permission use behaviors. This experiment clearly demonstrates that using permissions to automatically and systematically capture application behaviors is superior to traditional simple taint analysis without permission in consideration.

Finding 2: *VetDroid* can inspect the fine-grained causes of information leakage. Our permission use behavior 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 context of information leakage, we find that many such information leaks are actually not caused by the app itself. Table 4 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.

Compared with TaintDroid that could only alert information leaks, the results show that *VetDroid* is capable of inspecting the fine-grained causes of sensitive information leakage by tracing the context of permission usage.

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 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. This kind of *Account Hijack attack* could cause the same damage as *Facebook Account Hijack* [4].

We use *VetDroid* to reconstruct the permission use behavior of the activation process, as shown in Figure 8. As this figure 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 permission use behavior in Figure 8, it is easy to find that Viber does not check the origin of an activation SMS. Thus, an attacker could pass the activation by

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

Table 5: Results of execution time and memory footprint overhead on CaffeineMark benchmark.

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* [3], *Zitmo* [10] 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. Interestingly, in a security study [52] that performed network traffic analysis of nine popular VoIP apps, Viber was considered to be immune from *Account Hijack attacks*, because the activation code was generated in the Viber server and thus cannot be hijacked by a man-in-the-middle attack. However, our internal permission use behavior analysis on Viber reveals that the missing check on the origin of activation SMS actually makes Viber vulnerable to *Account Hijack attack*.

4.3 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 5 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 [24]. We believe this is a very reasonable and acceptable overhead for an offline analysis tool.

To measure the performance penalty in the worst case, we also write a benchmark app that invokes a privileged Android API 10,000 times and opens a socket 10,000 times. This case is used to measure the pure overhead caused by our permission check identification module. By measuring the execution time, we find the identification of *AndPermChk*s and *KerPermChk*s incurs an overhead of 80.108% and 238.870%, respectively. As the execution time of privileged calls represent only a small portion of the whole execution, *VetDroid* is quite efficient, especially as an offline analysis tool.

5. DISCUSSION

In this paper we focus on providing a new perspective for analyzing Android apps. 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 [48], AppInspector [34], Dynodroid [43], or guided analysis technique such as multi-path exploration [44], forced/informed execution [54, 55].

Our *I-PUP Tracker* is built upon TaintDroid, thus inheriting similar limitations of TaintDroid such as incapable of tracking native code and implicit flows, which we leave as our future work. One possible way to solve the native code problem is to build a taint analysis system that seamlessly tracks Java code and native code. In addition, implicit flows can also be accurately tracked by selectively propagating tainted control dependencies as in DTA++ [38]. 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 [35].

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 Behavior* provides a better approximation of sensitive behaviors inside an Android app. Thus, we believe *VetDroid* could be integrated with existing behavior-based malware detection techniques [14, 39] to extract effective malware signatures for clustering and detection.

6. RELATED WORK

Malware Analysis. As mentioned before, syscall-based solutions (e.g., syscall vector [16], syscall sequence with arguments [49], temporal pattern of syscalls [14], resource access model [40], syscall dependency graph [21, 22, 33]) are not well-suited for the Android platform due to the inability of monitoring Android-specific behaviors. DroidScope [56] 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 [59] and [29]. DroidRanger [60] and RiskRanker [36] 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*.

To analyze apps at market-scale, Chakradeo et al. [18] proposed app triage to efficiently allocate malware analysis resources. *VetDroid* can be combined with this technique into a practical market-scale application analysis solution.

Permission Analysis. Felt et al. [30] studied the effectiveness of the time-of-use and install-time permission grant mechanism. This work was extended in [28] 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 [25] to design a lightweight certification framework that could mitigate malware at install time. Apex [45] and Saint [46] were two extensions to the Android's permission system by introducing runtime constraints on the granted permissions.

To help end users understand application behaviors at install time, AppProfiler [50] 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.

Barrera et al. [13] 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. [27] 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 [32] or permission request sets [47] were also used to indicate the risk of new applications. To extract permission specifications for Android, Stowaway [27] used API fuzz testing while PScout [12] 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 [23, 31]. Grace et al. [35] empirically evaluated the re-delegated permission leaks in pre-installed apps of stock Android smartphones. CHEX [41] and DroidChecker [19] were two tools that could detect such kind of capability leaks. Bugiel et al. [15] 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. [20] 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.

Our *VetDroid* differs from all existing work in that it provides the first systematic framework to analyze permission use behaviors.

7. 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.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments and feedback. This work is funded by National Natural Science Foundation of China (NO. 61103078, 61300027), Science and Technology Commission of Shanghai Municipality (NO. 11DZ2281500, 11511504404, 13511504402 and 13JC1400800), a joint program between China Ministry of Education and Intel numbered MOE-INTEL201202, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project numbered B114. This work is also partially supported by the National Science Foundation under Grant no. CNS-0954096.

9. REFERENCES

[1] Android permissions. http://developer.android.com/reference/android/Manifest.permission.html.

[2] Androidos.tapsnake: Watching your every move. http://www.symantec.com/connect/blogs/androidostapsnake-watching-your-every-move.

[3] Android.smsreplicator. http://www.symantec.com/security_response/writeup.jsp?docid=2010-110214-1252-99.

[4] Facebook security phishing attack in the wild. http://www.securelist.com/en/blog/208193325/Facebook_Security_Phishing_Attack_In_The_Wild.

[5] Idc: Android market share reached 75% worldwide in q3 2012. http://techcrunch.com/2012/11/02/idc-android-market-share-reached-75-worldwide-in-q3-2012/.

[6] Mcafee threats report: Third quarter 2012. http://www.mcafee.com/ca/resources/reports/rp-quarterly-threat-q3-2012.pdf.

[7] Pendingintent. http://developer.android.com/reference/android/app/PendingIntent.html.

[8] Sms emulation using the android emulator. http://developer.android.com/tools/devices/emulator.html#sms.

[9] Ui/application exerciser monkey. http://developer.android.com/tools/help/monkey.html.

[10] Zeus-in-the-mobile - facts and theories. http://www.securelist.com/en/analysis/204792194/ZeuS_in_the_Mobile_Facts_and_Theories.

[11] Apple: ios 4. http://www.apple.com/iphone, 2011.

[12] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proc. of ACM CCS'12*, 2012.

[13] 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. of ACM CCS'10*, 2010.

[14] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proc. of MobiSys'08*, 2008.

[15] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proc. of NDSS'12*, Feb. 2012.

[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proc. of SPSM'11*, 2011.

[17] 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. of ISSTA'12*, 2012.

[18] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: triage for market-scale mobile malware analysis. In *Proc. of WiSec'13*, 2013.

[19] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proc. of WiSec'12*, 2012.

[20] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song. Contextual policy enforcement in android applications with permission event graphs. In *Proc. of NDSS'13*, February 2013.

[21] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proc. of ESEC-FSE'07*, 2007.

[22] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs. In *Proc. of IEEE S&P'10*, 2010.

[23] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: lightweight provenance for smart phone operating systems. In *Proc. of USENIX Security'11*, 2011.

[24] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of OSDI'10*, 2010.

[25] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of ACM CCS'09*, 2009.

[26] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, Jan. 2009.

[27] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of ACM CCS'11*, 2011.

[28] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *Proc. of HotSec'12*, 2012.

[29] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proc. of SPSM'11*, 2011.

[30] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proc. of WebApps'11*, 2011.

[31] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: attacks and defenses. In *Proc. of USENIX Security'11*, 2011.

[32] M. Frank, B. Dong, A. P. Felt, and D. Song. Mining permission request patterns from android and facebook applications. In *Proc. of ICDM'12*, 2012.

[33] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proc. of IEEE S&P'10*, 2010.

[34] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proc. of 2nd international workshop on Mobile cloud computing and services (MCS'11)*, 2011.

[35] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proc. of NDSS'12*, 2012.

[36] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of MobiSys'12*, 2012.

[37] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. of WiSec'12*, 2012.

[38] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. of NDSS'11*, Feb. 2011.

[39] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. of USENIX Security'09*, 2009.

[40] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proc. of ACM CCS'10*, 2010.

[41] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Prof. of ACM CCS'12*, 2012.

[42] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu. Shadow attacks: Automatically evading system-call-behavior based malware detection. *Springer Journal in Computer Virology*, 2012.

[43] A. MacHiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. Technical report, Program Analysis Group, Georgia Tech, 2012.

[44] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. of IEEE S&P'07*, 2007.

[45] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proc. of AsiaCCS'10*, 2010.

[46] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Proc. of ACSAC'09*, 2009.

[47] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proc. of ACM CCS'12*, 2012.

[48] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proc. of CODASPY'13*, 2013.

[49] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Proc. of DIMVA'08*, 2008.

[50] 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. of CODASPY'13*, 2013.

[51] H.-G. Schmidt, K. Raddatz, A.-D. Schmidt, A. Camtepe, and S. Albayrak. Google android: A comprehensive introduction. Technical report, DAI-Labor, TU Berlin, 2009.

[52] S. Schrittwieser, P. Fruehwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. R. Weippl. Guess who is texting you? evaluating the security of smartphone messaging applications. In *Prof. of NDSS'12*, Feb 2012.

[53] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proc. of Mobicom'12*, 2012.

[54] J. Wilhelm and T.-c. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proc. of RAID'07*, 2007.

[55] Z. Xu, L. Chen, G. Gu, and C. Kruegel. Peerpress: utilizing enemies' p2p strength against them. In *Proc. of ACM CCS'12*, 2012.

[56] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proc. of USENIX Security'12*, 2012.

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

[58] H. R. Zeidanloo and A. A. Manaf. Botnet command and control mechanisms. In *Proc. of ICCEE'09*, 2009.

[59] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of IEEE S&P'12*, 2012.

[60] 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. of NDSS'12*, 2012.