Compac: Enforce Component-Level Access Control in Android *

Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du
Dept. of Electrical Engineering & Computer Science
Syracuse University, Syracuse, New York
{ywang123,srhariha,chzhao,jliu20,wedu}@syr.edu

ABSTRACT

In Android applications, third-party components may bring potential security problems, because they have the same privilege as the applications but cannot be fully trusted. It is desirable if their privileges can be restricted. To minimize the privilege of the third-party components, we develop Compac to achieve a fine-grained access control at application's component level. Compac allows developers and users to assign a subset of an application's permissions to some of the application's components. By leveraging the runtime Java package information, the system can acquire the component information that is running in the application. After that, the system makes decisions on privileged access requests according to the policy defined by the developer and user. We have implemented the prototype in Android 4.0.4, and have conducted a comprehensive evaluation. Our case studies show that Compac can effectively restrict the third-party components' permissions. Antutu benchmark shows that the overall score of our work achieves 97.4%, compared with the score of the original Android. In conclusion, Compac can mitigate the damage caused by third-party components with ignorable overhead.

1. INTRODUCTION

Android has become the most popular smartphone platform, taking more than 70 percent of the market shares [3]. Android uses permissions to restrict the behaviors of apps. An app (In the rest of the paper, apps are used for Android applications) needs to have specific permissions in order to access protected resources. The app declares permissions that it needs in `AndroidManifest.xml`. During app installation, users are asked to approve the declared permissions. Upon approval, the app will be installed.

In Android, permissions are assigned at the app level. When an app is installed, it is assigned a unique UID, and each UID is associated with a set of permissions. At runtime, access control uses the UID to find out the permissions of an app, regardless of what component of the app is making the access. Therefore, all the components in the same app have exactly the same permissions. This is not a problem if all these components come from the same developer. However, this is not the case in Android and most other mobile systems. In these systems, apps often include third-party components. In-app advertisement is the most representative example. When an app needs to display ads, it has to incorporate the advertisement code (e.g. Google's AdMob). Once incorporated, both the ads and the original app will have the same privilege. Other commonly used third-party components include social networking service APIs, PhoneGap plug-ins, etc.

In most situations, apps need more permissions than what each component needs; when users grant the permissions to the apps, they also grant the same permissions to the components, leading to over-privileged components. In this case, some components in apps have more privilege than what they need. If they are malicious or have security flaws, they can cause problems. Previous work [44] indicates that several third-party components abuse apps' permissions to collect users' private information, without user consent.

Several ideas, such as AdDroid [34] and AdSplit [36], have been proposed to address the problem caused by a particular type of third-party component, namely, advertisement. AdSplit proposes to put ads in another process, isolating them from the app. AdDroid proposes to put ads into a service and assign a new ADVERTISING or LOCATION_ADVERTISING permission to this service. However, these solutions are mainly designed for advertisement; they are not general enough to extend to other types of third-party components.

In this paper, we propose COMPAC (COMPonent Access Control), a more generic solution, which extends Android's UID-based permission model [5] into UID- and component-based permission model. More specifically, several untrusted Java packages can be grouped into components, and an app can be divided into different components; app developer assign permission sets for the app and its components in `AndroidManifest.xml`. In such a case, both the app and components' permission sets are subject to Android system's control. To the best of our knowledge, this is the first time to propose a generic solution to limit part of app code's privileges in Android.

We have implemented Compac in Android 4.0.4, and have conducted a comprehensive evaluation on it, not only on its performance, but more importantly, on how Compac can be used to solve the over-privilege problems faced by today's

*This work is supported in part by NSF grants No. 1017771, No. 1318814, and by a Google research award.

apps when they use third-party components. Our results show that Compac is effective and applicable. We summarize the main contributions of this work as follows:

1. To restrict the privilege of third-party components, we propose a generic solution to contain in-app components' privileges by leveraging Java source code and extending the current Android permission model.

2. We implement the prototype called Compac and conduct comprehensive studies and performance evaluation to demonstrate that our approach can mitigate the damage caused by third-party code.

3. We provide app developers Compac APIs encapsulated in a customized SDK, which is compatible with the original Android SDK. We also develop a component permission manager to allow users to control app components' permissions on their devices.

2. BACKGROUND: ACCESS CONTROL IN ANDROID

In this section, we introduce the android sandbox protection for the system and data as well as the access control for non-privilege apps. At last, we define the reference monitors that will be used in our work.

2.1 Android Sandboxing

Generally speaking, Android has two types of resources that need to be protected. One is the app resources including the processes and files. The other is the system resources, such as camera, network, radio, GPS, and various sensors. To protect these resources, Android has developed a sandboxing mechanism that prevents apps from accessing the system resources or each other's resources. There are two aspects in this protection. First, each app is assigned a unique Linux UID, so Android can use Linux's UID-based DAC to restrict the privilege of apps. This naturally achieves the isolation among apps. Second, Android assigns most of the system resources to `system` UID, so they cannot be accessed directly by non-privilege apps.

2.2 The "Windows" on the Sandbox

Obviously, merely having such a sandbox is too restrictive. Apps should be allowed to access the system resources, as well as each other's resources. However, such accesses should be controlled, not arbitrary. The sandboxing mechanism eliminates the arbitrary accesses. To allow controlled accesses, "windows" have to be opened on the sandbox, but behind each window, there should be an access control.

System Calls. System calls are a typical way to allow user-level programs to access kernel-level resources. Android uses system calls to access some of the protected resources. For example, accessing the Internet is done through system calls, i.e., only apps with `inet` GID can directly access these resources by making the corresponding system calls. When an app invokes a system call to create an inet socket, the system call checks if the app has the `inet` GID; if it has, the app can get the socket and be able to access the Internet.

Android Permissions. Behind the GID check in privileged system calls, there are Android permissions, making sure that the authorized non-privilege apps can access the intended resources. Take the INTERNET permission as an example. When an app with the INTERNET permission is installed, Android will assign the `inet` GID to the app and its processes, as one of its additional GIDs. When the app accesses the Internet, request will be granted because of the `inet` GID; Unlike kernel resources, Android framework resources including services, content providers and broadcasts cannot be accessed directly by the system calls. These system resources provide an Android IPC interfaces as the "windows" to normal apps and Android permissions access control is built behind each window. When an app with the required permissions tries to access the intended system resources. The system resources call framework reference monitor to check the caller's permissions.

2.3 Reference Monitors

In the "windows" mentioned above, their access controls all have to get to one point: does the app have a particular permission? The windows themselves do not know the answer, they have to ask Android for its decision. In Android, this decision is only made in two places: at the framework level or at the kernel. We call them Framework Reference Monitor (FRM) or Kernel Reference Monitor (KRM).

FRM resides in the `system_server` process and it consists of two system services: Activity Manager Service (AMS) and Package Manager Service (PMS). Activities, content providers, services, and broadcasts check permissions using FRM. Whenever they need permission check, they send an IPC to AMS, which works together with PMS to conduct the check. In some cases, the caller is already in the `system_server` process, so the call will be a local one, not an IPC.

KRM resides in the kernel. Conceptually, the Linux Discretionary Access Control (DAC) is considered as KRM. When conducting permission check, KRM cannot reach out to AMS and PMS for the app's permission information. Pushing the permission information into the kernel can solve the problem, but can introduce significant kernel-level modification. Android chooses to utilize the Linux DAC by leveraging the UID and GIDs of a process to make the access control decisions in the kernel. For example, accesses to bluetooth, sdcard and the Internet need GIDs of `net_bt`, `sdcard_r` and `inet` respectively.

3. THE COMPAC DESIGN

3.1 The Overview

The main objective of Compac is to provide component-level access control. In Compac, each app consists of one or more components[1]. App developers or device users can grant different permissions to components. For example, an app has two components, one for the main activity, and the other for advertisements. The app can give only one permission (INTERNET) to the advertisement components, while assigning several permissions (such as READ_PHONE_STATE, ACCESS_COARSE_LOCATION) to the rest of the app. Such a fine-grained access control is not possible in the existing Android system: each app can have one set of permissions;

[1]Although Android platform has its own Android component definition, we use components to indicate Java packages. For example, Google Ads component means Google Ads related packages. Component is an abstract concept to facilitate the understanding of Compac.

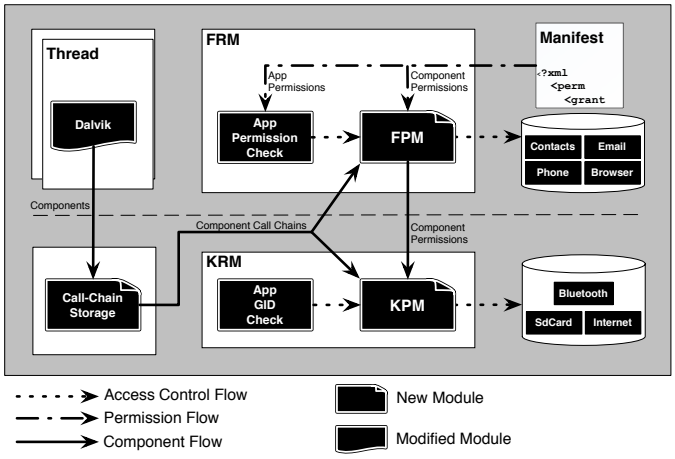


Figure 1: The Architecture of Compac

once granted by the users, all the components in the app will have the same privilege.

The architecture of our design is depicted in Figure 1. It is composed of three pieces. First, an app developer defines the app's permissions as well as the component permissions. When the app is installed, its app and component permissions are acquired by the reference monitors. As we mentioned in Section 2, Android has two permission-checking places: Framework Reference Monitor (FRM) and Kernel Reference Monitor (KRM). FRM still stores the app permissions as usual, and we do not modify the computing logic of the app permissions. To keep the best compatibility with current Android access control architecture, we extend the two RMs by adding two Policy Managers (PMs): Framework Policy Manager (FPM) and Kernel Policy Manager (KPM). The two PMs hold the component permissions that are obtained from the app's `AndroidManifest.xml`. Whenever permission update happens in Android framework, FPM synchronizes component permissions with KPM.

The second part is to extract the component information (Java package call chain) at runtime. We build hooks in Dalvik to trace the realtime Java method invocations. The trace is a call chain, which works like a call stack (FIFO). In order to make sure the component information is accurately recored, each thread of the app process has a call stack. If the thread is suspended, its call stack is locked. The thread's call stack follows the basic security rule. When a thread is forked from its parent, the parent's call stack status is inherited to avoid privilege escalation.

The third piece is access control enforcement. The RMs make decisions based on app's UID permissions and component permissions. RMs first check if the app has the requested permissions. If it does, RMs ask PMs to check the component permissions. PMs consider a call chain as the principal in an access control. They have the privilege to request the call chain from Dalvik. After the PMs finish the computation of permissions, they check the component policy to return the result to RMs; If the app doesn't have the requested permission, the request is denied without any further component permission check.

3.2 Assumption & Trusted Computing Base

Most Android apps are written in Java, but for performance reasons, Android allows apps to include native code (compiled from C/C++ code) [4]. To distinguish this type of native code from that provided by the Android OS, we call it the app-specific native code. Since this native code runs in the same process space as Dalvik, if it is malicious, it can tamper with the process' stack and heap memory, including the memory used by Dalvik.

In Compac, Dalvik provides component call-stack information. If Dalvik's data memory can be changed by malicious components (through native code), there is no guarantee on the integrity of the runtime component information. In this paper, because of the lack of isolation between the app-specific native code and Dalvik, we block the invocation of app-specific native code in apps. The assumption is only temporary and has limited impact:

1. Based on the previous study [45], only 4.75% of benign apps have native code. Therefore, the majority of the apps will not be affected by this assumption.

2. Isolating the app-specific native code from the rest of the system is not impossible to implement. This goal is already achieved in the Chrome browser by the Native Client (NaCl) framework [43] using Software Fault Isolation (SFI) [40]. Robusta [37, 39] has successfully isolated the native code from Java Virtual Machine (JVM) in the traditional OSes. It will be just a matter of time before the isolation of the native code from Dalvik is achieved in Android.

3. If a component's permission set is the same as the app's permission set, i.e., there is no permission restriction on this component, we do allow the native code to be invoked by this component, because no extra privilege can be gained by this component even if it can modify Dalvik's memory. The blocking of native code is only enforced if a component has less permissions than the app.

It is quite tempting to enforce the component-level access control inside Dalvik, just like what the security manager does in the traditional JVM's security architecture [10]. In Android, the existing access control is not enforced inside Dalvik; instead, it is enforced either inside the kernel or in a privileged process (i.e., `system_server`). Android chooses to conduct access control in this way, rather than simply using the security architecture of JVM; this is mostly because Android itself uses a great deal of native code, in addition to the native code brought by apps. When an app invokes the native code, Dalvik will have no control. Thus we enforce the access control policy in Android framework not Dalvik. However, in order to secure component boundaries and ensure the correct runtime component information, we do implement some auxiliary security rules (see section 4.2) in Dalivk. In general, we only use Dalvik to get the runtime component information, not for access control.

3.3 Component Permission Configuration

In the original Android platform, apps need to specify all the permissions they need in `AndroidManifest.xml`. During the installation, users will be asked whether they want to grant the requested permissions. Once granted the permissions, an app will have those permissions until it is uninstalled [25]. When the app requests protected resources, the corresponding permission will be checked.

To support component-level access control, we need to attach a separate permission set to different components. Consider all the participants involved in app management, We provide component permission configuration in two different ways. First, an app developer can specify what permissions each component needs. This is done by adding a special section in `AndroidManifest.xml`[2]:

```
<uses-permission android:name="READ_CONTACTS
    "/>
<uses-permission android:name="INTERNET"/>
<package-permission android:name="com.google.
    ads">
  <assign-permission android:name="INTERNET"/>
</package-permission>
```

In the above permission definition, the app's permissions are defined in Android's original `uses-permission` tag. The app has the INTERNET and READ_CONTACTS permissions. We call these permissions the app's default permissions. For the components that are not specifically mentioned in `AndroidManifest.xml`, they will have app's default permissions. If app developers want to restrict the permissions of some third-party components, they need to specify that using our new tag called `package-permission`. In the example, we have assigned only the INTERNET permission to the `com.google.ads` component. This component can use the Internet, but will not be able to read the user's contact data.

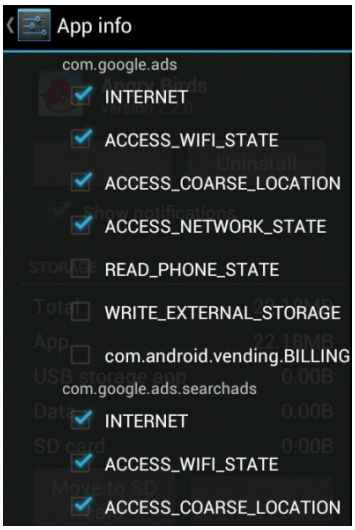


Figure 2: Permission Manager Interface for Users: third-party components with less permissions will be displayed in system settings, and experienced users have options to adjust component permissions in the app info section of system settings.

Second, experienced users have options to set their own security policies on a component, putting a restriction on what permissions it can have (see Figure 2), regardless of what are assigned to the component during the installation. This is very useful for users to control the popular components, such as advertising components, social networking service APIs, etc. Meanwhile, inexperienced users can keep apps' default settings configured by the developers. Users set component permissions per app. For example, if a user assigns only the INTERNET permission to the `AdMob` package in one app, then it will not affect the other apps' `AdMob` component permissions. This setting only overwrites the permissions assigned to the component in this app.

All the app permission definitions will be stored in framework reference monitor as what original Android does. However, all the component permission definitions will be acquired by framework policy manager. Once any component permission update happens, framework policy manager synchronizes the information of the component and its permissions with kernel policy manager. The synchronization is only done when the app is installed/updated, or when users modify the permissions assigned to a component.

3.4 Extracting Component Call Chains

To enforce component-level access control, Android needs to know what component initiates the access in addition to the UID information. However, that is not enough, as one component A can invoke another component B, and B then invokes C, which initiates the access. To make a correct access control decision, Android needs to know the entire call chain $A \rightarrow B \rightarrow C$, instead of C alone. This call chain is called the component call chain in this paper. Since Dalvik supports multi-thread, the call chain must be extracted at the thread level, one per thread. When a new thread is spawned, its initial call chain is inherited from the parent thread. The new thread has to keep the initial call chain during its lifetime to avoid escalating its privilege.

The component call chain needs to be extracted at runtime from inside Dalvik. Dalvik functions as the Java interpreter in Android; it converts Java bytecode in dex format into native code and executes the native code [23]. To extract the call chain, we put hooks in Dalvik.

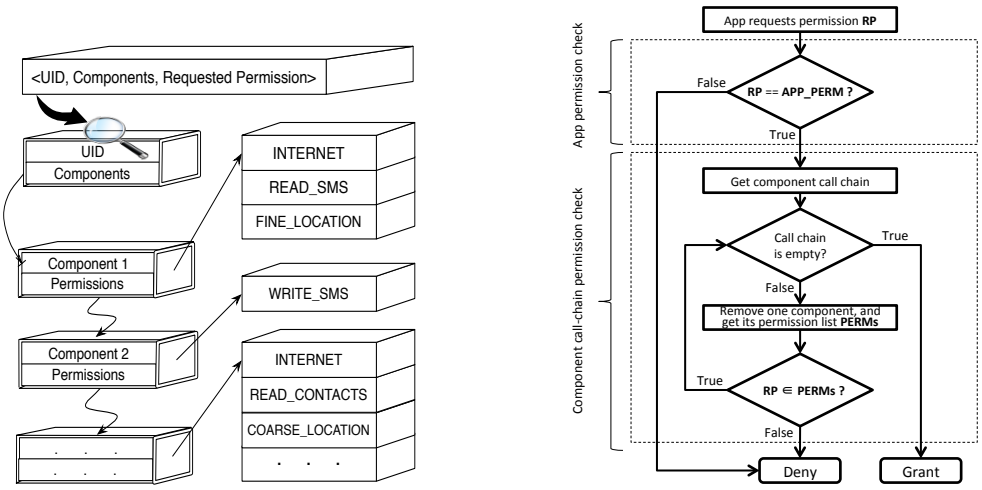
3.4.1 Hooks

Dalvik's core interpreter is called `mterp` [42], which interprets the machine-independent bytecode to machine-dependent code. There are many opcodes in bytecode, such as conditional, mathematical, method operations, etc., but component transitions (i.e., from one component to another component) can only happen at the method invocation and return time, so we only focus on the opcodes related to method invocation and return. We have identified all these opcodes [3], and placed a corresponding hook in each of them. These hooks check whether there is a component transition; if so, record the transition to the call chain.

In addition to the hooks placed in `mterp`, we also place hooks in the Java reflection class and DexClassLoader to address implicit method invocation and code injection attacks. We will further discuss these two attacks in Section 4.

[2]For formatting reasons, permission names in our examples are shortened by removing their prefix. For example, the full name of READ_CONTACTS should be android.permission.READ_CONTACTS.

[3]We place hooks in the following opcodes: invoke-virtual, invoke-super, invoke-direct, invoke-static, invoke-interface, invoke-virtual/range,invoke-super/range, invoke-direct/range, invoke-static/range, invoke-interface-range, invoke-virtual-quick, invoke-virtual-quick/range, invoke-super-quick, invoke-super-quick/range, return-void, return vx, return-wide vx, and return-object vx.



(a) Policy Manager Structure (b) Permission Checking Flow

Figure 3: Policy Manager (PM) Structure & Permission Checking Flow

3.4.2 Call-chain storage

Once call chains are collected, the first question is where to store them. With respect to call chain extraction and permission enforcement, three candidate places are safe to store call chains: Dalvik, the kernel and the `system_server` process. From cost efficiency perspective, the kernel is the best place to store call chains. Suppose call chains are stored in Dalvik, every time `system_server` checks permissions, `system_server` needs to request call chain through IPCs. Similarly, if call chains are stored in `system_server`, whenever Dalvik updates a call chain, it has to talk to `system_server` via IPCs. These two operations are relatively frequent, and so many IPCs will lead to low performance. By storing call chains in the kernel, these two kinds of IPCs turn to system calls. According to our experiments, we find storing call chains in the kernel boosts the speed of call-chain update and permission checks.

Compac stores call chain based on thread. A process can have several threads running simultaneously, thus can have multiple component call chains, one for each thread. When Dalvik detects a component change, it sends the new component information into the kernel, or asks the kernel to remove one from its call chain, depending on whether it is an invocation or return.

Recording a call chain in Dalvik may introduce a high overhead if it is not properly handled. For performance reasons, we optimize the design in three aspects: first, we do not record all component transitions. We only record transitions among the components specified in the `AndroidManifest.xml` and the user's settings, because only these components can cause permission changes. Second, we delay the call-chain synchronization at Java Native Interface (JNI). Dalvik does not synchronize call chain with the kernel every time when there is a component (package) transition. Instead, it does that in JNI. This is because the Java code will eventually be interpreted to native code provided by Android, and JNI is the only entry point where the transition from Java code to native code happens. Third, method invocation within the same component will not be recorded. For example, if a method in package A invokes another method in the same package, which in turns invokes a method in package B, the call chain will be "$A \rightarrow B$".

3.5 Access Control Enforcement Based on the Call Chain

Compac's permission model is composed of app permission check and component permission check (see Figure 3(b)). Compac keeps a clear and standalone design for the two permission checks. Compac remains the app permission check logic in RMs without modification, meanwhile, Compac has two new modules called *policy managers* and *kernel policy manager* to check the component permissions. The two Policy Managers (PMs) are built in the two Reference Monitors (RMs) separately. When a permission request comes to RMs, RMs first check app permissions. If the app does not have the specific permission, the request is denied immediately without going to PMs. Otherwise, RMs ask PMs to begin the component permission checking procedure. PMs send a privileged system call to call-chain storage in order to request component call chain. Once they receive the call chain, they calculate the call-chain permissions and check whether the caller's call chain have the requested permission. PMs return the result to RMs, no matter what the result is. After that, RMs handle the rest as what they do in original Android.

We name the permissions calculated from component call-chain permissions as the *effective permissions*. In the original Android framework, there is no such call chain, so the effective permissions are the same as the app's permissions granted during the installation. In Compac, the effective permissions are calculated as the following definition:

Definition *Effective Permission.* Let $C_1, \ldots, C_n$ be components, and P_i be the permission set for the component C_i, where $i = 1, \ldots, n$. Assume that the current component call chain for a thread is $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n$. The effective permission set eP of the current thread is defined as the following:

$$eP = P_1 \cap P_2 \cap \ldots \cap P_n.$$

If $P_{request} \in eP$, PMs allow the access request. In the ac-

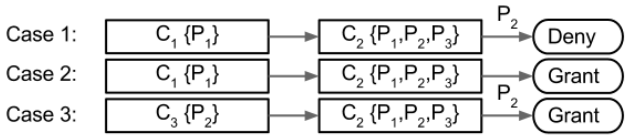


Figure 4: Component Intersection Cases

tual enforcement, PMs do not calculate all the permissions, instead, PMs examine the call chain and just check each component to see if they have the requested permission.

The policy seems too restricted, however, it is effective and practical. First, all the components' permissions of an app are defined by one app developer. Consider the example in Figure 4, there are three components C_1, C_2, C_3, and they have permission sets $\{P_1\}$, $\{P_1, P_2, P_3\}$ and $\{P_2\}$. In case 1, when C_1 calls C_2 in order to use P_2, the access is denied. If the developer would like C_1 to use P_2, the developer will assign P_2 to C_1 just like C_3 in case 3. Otherwise, C_1 is trying to gain P_2 without user consent. So, the deny decision is correct. In case 2, we can see the policy allows the components with less privileges to call components with more privileges. As long as the privileged action is not requested, PMs will not check component permissions. Second, the call chain only records components (packages) that are specified with tag `package-permission` in `AndroidManifest.xml`. These are untrusted third-party libraries and there are not too many cross references among them. We conduct several experiments on Compac by recording third party components and logging the call chains. We never see the call chain having more than five components. We also evaluate 34 apps (downloaded from Google Play) by restricting all the third-party components, and we never experience false positives.

3.6 Policy Manager Implementation

Policy managers play a critical role in component call-chain access control. Thus, we take FPM as an example to illustrate the internal work flow in PMs. After app permission request is granted, the permission request comes to FPM and FPM begins to check component call-chain permissions. First, FPM gets the caller's identity from Binder IPC, In this case, the caller's identity is thread ID (TID). With the TID, FPM requests and gets the thread's component call chain from call-chain storage. Besides, FPM also has the caller's UID, so FPM can map out the app's components with their permissions from FPM's internal three-level hierarchical data structure, as shown in Figure 3(a). KPM has a similar checking procedure except that KPM gets the call chain much easier, because there is no IPC and context switch involved.

During the FPM permission checking procedure, the most difficulty is how the FPM gets the caller's TID since there is no TID related support in original Android. It is much easier if the caller voluntarily sends the TID to FPM. However, such information cannot be trusted. We choose to let the kernel provide the caller's TID. We allow the FPM to call `getCallingTid` API, which is similar to the system's `getCallingUid` API. FPM can use `getCallingUid` to request UID from binder's kernel driver when FPM needs UID to check app permission. In our implementation, we add the `getCallingTid` API support in Binder's kernel driver by recording the caller's TID when IPC happens. We also build both the Java and native `getCallingTid` interfaces for any IPC that goes through binder. As a result, FPM can get the caller's TID through binder object for authorization.

4. POTENTIAL ATTACKS & DEFENSE

The traditional JVM has its own built-in access control [27], which is enforced through the security manager, access controller, and security packages in `java.security.*`. This access control in general falls into two categories: the access control on resources (such as file read/write, hardware access, etc.) and the access control on language properties (such as whether or not a program can use Java reflection or class loader).

To contain native code, Android relies on reference monitors to protect privileged resources and removes access control entirely from Dalvik. Without this kind of access control, it is difficult to achieve the isolation among components. For example, reflection can be used to inject code in another Java package, blurring the boundaries among components. Compac heavily relies on components, so we need to clearly identify component boundaries. In this section, we describe possible attacks on Compac and explain how we remedy these attacks in our design. In a nutshell, we need to compensate for the missing language security feature in Dalvik, in order to secure the component-level access control.

4.1 Implicit Invocation Attack

Attack. Reflection is a powerful Java language feature, which allows a piece of code to invoke any method of a class, including its private and protected methods, unless the method is marked as inaccessible for reflection. Based on our study (Table 1) [4], we can see that around 60% of benign apps use reflection to implicitly invoke other methods. Some usages are made by the advertisement code included in apps. After excluding that factor, about 42.49% of benign apps use reflection. The widely used reflection functionality drives the needs to handle this type of special invocations instead of simply blocking them.

Table 1: Implicit Invocation Usage (Total App Samples: Benign 16000, Malware 2566)

Implicit Invocation App Sample		Number	Percentage
Benign Apps	including Ads	9577	59.86%
	excluding Ads	6798	42.49%
Malware Apps	including Ads	1377	53.66%
	excluding Ads	989	38.54%

Defense. Compac can easily solve the reflection problem. The Java code is interpreted in Dalvik and Dalvik can trace all the methods including reflection methods. So Dalvik knows which method will be invoked in a reflection invocation. Reflection's implicit invocation is implemented in the `reflection_native()` method, which resides in the core libraries of Dalvik. We place a hook in `reflection_native` to monitor which Java method will be invoked by reflection, and thus extract the Java package containing the real invoked method instead of reflection packages.

[4]Benign apps are downloaded from Google Play, and most malware apps are collected from Android malware genome project [2].

4.2 Inter-Component Code Injection Attacks

If a package can modify another package's code, it can break Compac, because a package with less privileges can simply inject its code into a package with more privileges. We call such an attack the *inter-component code injection attack*. The Java instrumentation class, `java.lang.instrument` [8], can be used to modify the contents of an existing class; fortunately, it has been disabled in Android SDK. Instead, Android develops its own instrumentation package, `android.app.Instrumentation` [11]. This package can be disabled in `AndroidManifest.xml`. It seems that we are safe, but unfortunately, in addition to instrumentation, there are two other ways to modify other class's code.

Attack 1. Reflection can be used to modify the field of a class object (see the following example):

```
import java.lang.reflect.Field;
Field field = classInstance.getClass().
    getDeclaredField(fieldName);
  /* Allow modification on the field */
field.setAccessible(true);
  /* Set the field to a new value */
field.set(classInstance, newValue);
```

The field itself can be an object of any type. If the object field is changed, the instance is modified. Moreover, if a class object is changed, the corresponding methods are changed accordingly. Initially, we decide to simply block `set()` method, but our study (Table 2) has shown that about 15.24% (12.03% if ads are excluded) of benign apps use the reflection in this way.

Table 2: Reflection for Code Injection Usage (Total App Samples: Benign 16000, Malware 2566)

Code Modification App Sample		Number	Percentage
Benign Apps	including Ads	2438	15.24%
	excluding Ads	1925	12.03%
Malware Apps	including Ads	56	2.18%
	excluding Ads	17	0.66%

Attack 2. The second attack is related to the class loader. If developers restrict an untrusted component's permissions, the untrusted component can escape the restriction using `DexClassLoader`. Using the class loader, an untrusted component can reload a class [28], and therefore can replace a more trustworthy component with its own malicious code. This completely defeats the component-level access control. Class reloading is not possible in JVM, as special permissions need to be granted to an app before it can load classes. Since Android's Dalvik removes this access control, class reloading becomes possible.

When using class loader `DexClassLoader` in Dalvik, the protected method `loadClass()` in Dalvik does check whether the class is already loaded or not; if it is, it will not reload the same class. However, this is only enforced inside `DexClassLoader`, so if a malicious component extends `DexClassLoader` and overrides `loadClass()`, it can successfully reload a class.

Defense for attacks. For both attacks, we enforce the following policy in Dalvik: if code in package A tries to modify/reload a class in package B, this action is only allowed if $P_B \subseteq P_A$, where P_A and P_B are the permission sets of package A and B, respectively. In other words, a component with less privileges cannot modify/reload a class (component) with more privileges, and thus cannot gain privilege.

4.3 Package Forgery "Attack"

Since Compac identifies components using the Java package name, a potential attack is to forge the package name, so the restriction on the package can be circumvented. However, this is not a feasible attack. When developers intend to include a third-party package in their apps, they have the responsibility to ensure the integrity of the package. For example, if they plan to include Google advertisements in their apps, they need to ensure that the AdMob SDK they use is indeed from Google.

5. CASE STUDIES & EVALUATION

In this section, we conduct two types of evaluations. First, we use three case studies (Ads APIs, social networking service APIs, and web apps) to demonstrate how the privileges of components can be restricted using our component-level access control. In all the case studies, we focus on the most representative permissions, such as INTERNET, READ/WRITE/SEND_SMS, READ/WRITE_CONTACTS, READ_PHONE_STATE, ACCESS_COARSE/FINE_LOCATION. Second, we evaluate the performance of Compac.

5.1 Advertising APIs

We would like to evaluate how Compac works with various advertising packages. Certain advertising APIs (like `InMobi`) may use user's phone state or location information for displaying more relevant Ads to the user. To protect clients' privacy, a developer can prevent the advertising components from using these permissions without harming the app functionality and changing the app's permission set.

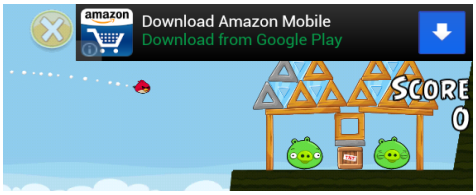


Figure 5: Angry Birds

We use the Angry Birds app [6] to demonstrate the aforementioned scenario. Angry Birds uses five advertisements including `AdMob`, `InMobi`, `Millenial Media`, `JumpTap` and `GreyStripe`. In Angry Birds, advertisements display at the top of the screen (see Figure 5), and the app randomly chooses one advertisement from the five to display. We assign only two necessary permissions for the five Ads APIs, while the Angry Birds app has six default permissions. Component permission tags are defined in `AndroidManifest.xml` (to save space, we only show the tags for Google Ads).

```
<package-permission android:name="com.google.
    ads">
 <assign-permission android:name="INTERNET" />
 <assign-permission android:name="
     ACCESS_NETWORK_STATE" />
</package-permission>
```

We repackage app's APK file and run the Angry Birds game. As the game runs, we suddenly receive a pop-up win-

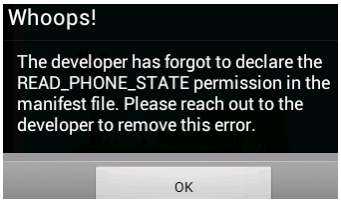


Figure 6: Angry Birds: No Permission Pop-up Window

```
D/dalvikvm(  663): DALVIKHOOK DEL pkg=com.jumptap.adtag; method=JtAdView.setAdViewListener|
StackSize=0
D/dalvikvm(  663): DALVIKHOOK ADD pkg=com.jumptap.adtag; method=JtAdView.loadUrlIfVisible|S
tackSize=1
D/PackageManager(   81):  checkUidPermissionTid android.permission.INTERNET is granted, uid
=10037, tid=663
D/PackageManager(   81):  checkUidPermissionTid android.permission.READ_PHONE_STATE is deni
ed, uid=10037, tid=663
E/JtAd    (  663): JtAdManager: Requires READ_PHONE_STATE permission
D/JtAd    (  663): Base url : http://a.jumptap.com/a/ads?textOnly=f&ua=Mozilla%2F5.0+%28Lin
ux%3B+U%3B+Android+4.0.4%3B+en-us%3B+Full+Android+on+Crespo%2FIMM76I%29+AppleWebKit%2F534.3
```

Figure 7: Ads in Angry Birds: Permission Deny Logcat

dow (Figure 6), indicating that the developer has not declared the READ_PHONE_STATE permission. From the logs (Figure 7), we can see that the `JumpTap` package throws an exception, but the game does not crash and continues running smoothly; this is because READ_PHONE_STATE is an optional permission and Ads handle it properly. Experienced users can achieve the same goal by modifying the package's permissions in the app setting.

Besides, we download 33 apps from Google Play and run the extreme experiments like giving empty permission set to an app' Ads. The results show that we can successfully restrict all the Ads in these apps. 29 apps handle the no-permission exception, so they continue to run without any problem, except that no advertisement is displayed. 4 apps display "no INTERNET permission" toast messages and then exit.

5.2 Social Networking Service APIs

Android apps use a number of social networking service (SNS) APIs, such as Facebook, Twitter, and Dropbox APIs. To use these APIs, apps can include the API packages in the program, and interact with the classes in the packages through the APIs. Once included, the API package will have the same privileges as the app. Unfortunately, some of the packages seem to abuse the privileges by collecting private information about users. It will be more desirable if apps can limit the privilege of these API packages.

The component-level access control in Compac can be used for this privilege-restriction purpose. To demonstrate that on SNS APIs, we emulate a scenario where a malicious SNS library attempts to read user contacts. We insert a small piece of malicious code in Facebook SDK. As long as the app that uses this API has the permission to read user's contacts, the inserted code can silently read the contacts from the phone and send them out, without user consent. We package the malicious Facebook SDK in a sample app. This app gets the information from the user's friend list (in Facebook), compares the list with the user's contacts (on the device), and sees whether any friend is on the contact list. This is a typical use of Facebook APIs. Without the protection from Compac, the privilege for reading contacts

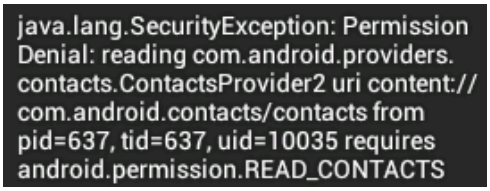


Figure 8: Facebook: Permission Deny Toast

```
D/ContentProvider(  231): enforceReadPermission uid=10035, tid=637
D/PackageManager(  79): checkUidPermissionTid android.permission.READ_CONTACTS is denied,
uid=10035, tid=637
E/DatabaseUtils(  231): java.lang.SecurityException: Permission Denial: reading com.androi
d.providers.contacts.ContactsProvider2 uri content://com.android.contacts/contacts from pi
d=637, tid=637, uid=10035 requires android.permission.READ_CONTACTS
```

Figure 9: Facebook: Permission Deny Logcat

may be abused by the malicious Facebook APIs.

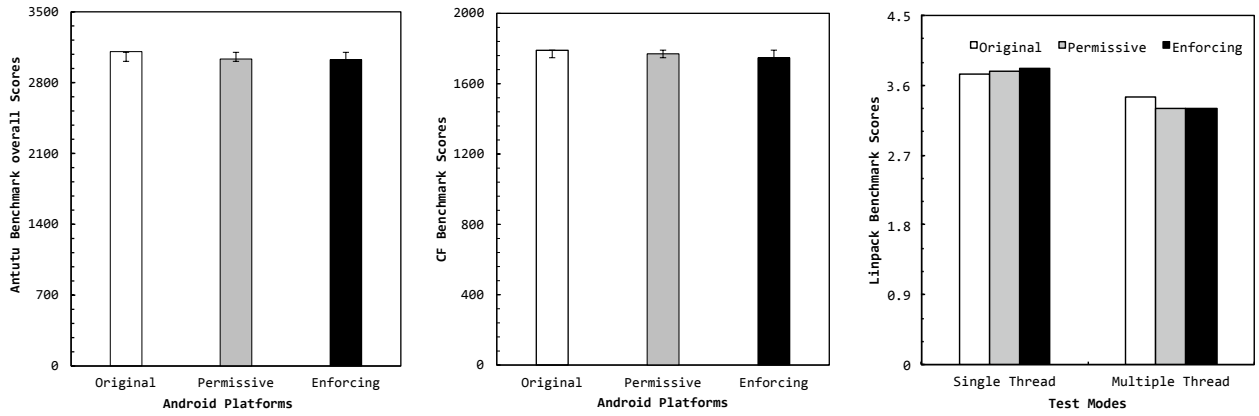
To block the READ_CONTACTS permission, we restrict the permissions of the Facebook SDK component to INTERNET only. When we run the app under Compac's protection, the app throws a "Permission Denial" toast for READ_CONTACTS (see Figure 8). As shown in Figure 9, the action (reading contacts from phone) has been blocked. The main app still has the READ_CONTACTS permission, so we can tell the blocked event is caused by the Facebook component.

5.3 Web Applications - PhoneGap

Compac can also protect web apps, although component hooks are not deployed in WebKit's JavaScript interpreter. To demonstrate that, we have conducted experiments on a popular cross-platform web app framework called PhoneGap [9]. PhoneGap encapsulates a WebView, allowing developers use HTML5, JavaScript, and CSS to develop mobile apps. The JavaScript APIs provided by PhoneGap can access device resources through WebView's `addJavascriptInterface` API [26]. However, this API has caused many security problems [31] because it is exposed without control. A recent article [1] demonstrates that JavaScript code can use reflection to gain the Java object reference, thus can gain the app's privilege. The security problem is because WebView cannot contain the JavaScript code.

Compac easily fixes the above over-privilege problem by assigning only the INTERNET permission to WebView component. Every time the `addJavascriptInterface` API uses reflection to access privileged Java API, it is recorded in the call-chain. When the privileged Java API tries to perform privileged actions on behalf of the JavaScript code, access is denied; If the Java API is called by the app's code and the `addJavascriptInterface` API is not in the call-chain, access is allowed.

Besides, Compac also provides a solution to enforce the principle of least privilege on PhoneGap plug-ins. In PhoneGap, a plug-in usually conducts a particular functionality, such as reading/sending SMS, using camera, etc. All plug-ins are included as libraries. Once included, they have the same privilege as the app, leading to an over-privilege problem. Since all the plug-ins have their own unique packages, third-party plug-ins can be considered as components. Based on the functionalities of plug-ins, we evaluate six different plug-ins including `com.seltzlab.mobile`, `com.leafcut.ctrac`, `com.rearden`, `org.devgeeks`,`com.karq.gbackup`, `com.practicaldeveloper.phonegap.plugins`. According to



(a) Antutu Benchmark (b) CF Benchmark (c) Linpack Benchmark

Figure 10: Overall Performance: Benchmark Results

their documents, they require permission GET_ACCOUNTS, CALL_LOG, READ_CONTACTS, READ_PHONE_STATE, READ_SMS, SEND_SMS individually.

We conduct two experiments on these plug-ins. In the first experiment, we remove all the permissions from the plug-ins. We have observed that Compac is able to limit the behaviors of five plug-ins and their requests to access the device resources are all denied, The only exception is the Contact View plug-in (`com.rearden`), which needs no permission and does not perform privilege action directly. Instead, it displays the contacts by sending Intent to Android's built-in Contacts app. This is a privilege escalation problem [15] between apps and out of this paper's discussion. In our second experiment, we assign only the required permission as their documents describe to the six plug-ins. Our results show that all the plug-ins work properly. This indicates that they do not need extra permissions. By combining the two experiments, we have demonstrated that Compac can resolve the over-privilege problem associated with the PhoneGap plug-ins, without affecting their functionalities.

In conclusion, we demonstrate that Compac can successfully put access control on WebView and other web frameworks, and this is not easily done by previous work.

5.4 Performance

The overhead of Compac mainly comes from two sources. One is from the hooks that we insert in Dalvik, and the other is from the component permission checks (at both framework and kernel levels). Component permission checks are more complicated, as the effective permissions need to be calculated from the component call chain at runtime. Dalvik hooks just record the necessary Java packages. So, the cost from permission checks is far more than that from Dalvik hooks. We first evaluate the permission check overhead under an extreme condition, which provides a guideline on the upper boundary of Compac's performance. Then we evaluate the overall performance, which illustrates the overhead in normal situations. We employ a Nexus S phone as the evaluation platform. Our codebase is the `Android-4.0.4_r1.2` branch from the Android Open Source Project (AOSP). The original Android and Android with Compac are originally from the same copy of the source code.

5.4.1 Permission Checking Overhead

We first measure the overhead of Compac's permission checks. In Android, As we know, permissions are checked either in the kernel through system calls, or at the framework level through IPC (IPC permission checks). The cost of an IPC permission check is always higher than a kernel permission check, because an IPC involves several system calls. Therefore, we only perform evaluation on IPC permission checks to measure the upper boundary of overhead caused by Compac's access control. We use the following pseudo-code to show how we conduct the unit test.

```
time_start := System.currentTimeMillis
for i:=1 to 25000 do
   check random permission
time_end := System.currentTimeMillis
excute_time := time_end - time_start
```

We do the test by conducting the IPC permission checks on random permissions 25000 times. On the original Android, the test takes 38,153 ms, and on Android with Compac, it takes 46,614 ms. Therefore, Compac's overhead for IPC permission checks is 22.2%. Although this number seems high, it just provides the upper boundary of the Compac's performance and does not pose problems for the overall performance. Because the IPC permission check happens only when a privileged action is triggered. A normal app does not have many such IPC permission checks.

5.4.2 Overall Performance

To evaluate Compac's overall performance, we use three popular benchmark apps: Antutu, CF-bench, and Linpack. We plot the results in Figure 10. In the figure, "Original" means the AOSP Android; "Permissive" means Android with Compac will be in permissive mode, i.e. it will log access control denials but not enforce them; "Enforcing" means we restrict each package's permissions inside the benchmark apps and Android with Compac runs in enforcing mode.

The three benchmark apps show similar results that the overall performance of Compac is quite close to the original Android platform. A higher number is better in all the three benchmarks. Take the Antutu benchmark app as an example. Antutu produces an overall score based on the measures. The result shows that the original Android score is 3106, Compac in permissive mode scores 3033, and Compac in enforcing mode scores 3026. Taking the original Android score as the base, the performance of Compac in permissive mode is 97.6% and the performance of Compac in enforcing

mode is 97.4%. One interesting thing is that for the single thread test with Linpack, Android with Compac has a higher score than original Android. We run the experiment several times and get similar results. Sometimes, Compac in permissive mode may score better than Compac in enforcing mode, but both always score better than original Android. The reason is we optimize Dalvik interpreter and related libraries when we implement Compac. According to Linpack for Android app's description [7], "the Dalvik VM has a huge impact on the Linpack number", and "this test is more a reflection of the state of the Android Dalvik Virtual Machine than of the floating point performance of the underlying processor. Software written for an Android device is written using Java code that the Dalvik VM interprets at run time".

6. RELATED WORK

Information Flow Tracking. To protect user privacy, the pioneering work TaintDroid [22] conducts taint analysis in Dalvik to trace information flows. It has the ability to trace when the sensitive information flows out of an app. TaintDroid taints data in Dalvik while our work builds hooks in Dalvik for system to make access control. AppFence [29] is built upon TaintDroid and can block unwanted data transmission.

Android Permission Control. A collection of work [19, 24, 32, 33, 35] adopt different policies to achieve fine-grained access control on app permissions in order to control an app's behaviors or make sure the app has the least privilege. Apex [32] and Kirin [24] allow users to accept a subset of the permissions declared by apps and enforce policies at installation-time. Saint [33] enforces policies at both installation-time and runtime, and the policies leverage the relationship between the caller app and the callee app. The user-driven work [35] considers certain user inputs as contexts. Upon occurrence of certain user actions in special UI components, the system grants corresponding permissions to the app. CRePE [19] uses environments as contexts to control the behavior of an app.

In-App Reference Monitor. Aurasium [41] and previous work [12, 20, 30] build a reference monitor within an app to achieve access control through code instrumentation or rewriting. They can flexibly enforce app level policies but may not enforce component-level policies. Because all the code as well as the reference monitor run in the same process. Without jumping out of Dalvik or the process, it is difficult for the reference monitor to recognize the running code. Besides, the current Android does not provide component information for access control either. However, Aurasium can still be used by app markets to protect users from malicious and untrusted apps if app markets wrap each app with a reference monitor and consider the whole app as the target.

Mandatory Access Control. Previous arts XManDroid [14], TrustDroid [16], IPC inspection [15] and Quire [21] deal with the privilege escalation problems across different apps, while Compac focuses on the components inside the same app (same process). They are the first work to propose mandatory access control concept in Android. SEDalvik [13] also intercepts the Java methods in Dalvik as Compac, but it enforces policy within Dalvik. SEDalvik can prevent some Java level malware, however, the system cannot be aware of the security contexts, and Dalvik cannot understand the contextual information of the code. So, SEDalvik turns to a language-level mandatory access control within apps.

SEAndroid [38] implements SELinux in Android kernel and builds middleware mandatory access control in both the kernel and Android framework. SEAndroid ports the core SELinux into Android's kernel and makes all the processes, files, sockets and other kernel resources under the control of the MAC. Thus, SEAndroid limits root and other users' privilege. SEAndroid's install-time MAC, intent MAC and content provider MAC provide a comprehensive protection to apps and Android middleware. SEAndroid and our work strictly follow the same design principle (least privilege principle). SEAndroid divides the privileges of the system, users and resources, while our work divides the privileges of the apps.

FlaskDroid [17, 18] extends the type enforcement policy language from SELinux to Android middleware and offers API-oriented MAC control for multiple stakeholders including app developers. FlaskDroid allows developers and users to have finer-grained access controls in services and content providers. For example, the content in content provider can be partially displayed according to the policy. FlaskDroid makes Android middleware be aware of the contextual information in services and content providers, and our work makes reference monitor get the component contexts of an app.

Component-level Access Control. AdDroid [34] isolates advertising components by encapsulating them as a service and creating ADVERTISING permission for the advertising service. AdSplit [36] enforces a fine-grained access control by putting the advertising component entirely in a separate process, which relies on the process-level isolation to achieve the protection. This solution implements a strong component isolation mainly for components like advertisements, which need less or no communication with other components. However, the solution is difficult to be extended to other types of components, such as PhoneGap, social networking service APIs, and WebView. Unlike advertisements, these components require close interactions with the apps. Putting them in another process will turn every interaction into an IPC, resulting in a significant overhead.

7. CONCLUSION

To reduce the risks caused by the untrusted third-party code included in Android apps, we propose Compac, a generic approach to achieve component-level access control. Compac extends the existing Android security model. It uses Java package as component and enforces access control based on the UID and component. Using Compac, a component in an app can be given a subset of the app's permissions. We conduct case studies on various third-party APIs including advertising, social networking service, PhoneGap plug-ins, and WebView to demonstrate its effectiveness, usefulness and the compatibility with the existing Android architecture. We also evaluate its performance to show that the overhead is quite affordable. The source code of Compac is available upon request for now, and will be released on GitHub to the public soon.

8. REFERENCES

[1] Abusing webview javascript bridges. http://50.56.33.56/blog/?m=201212.

[2] Android malware genome project. http://www.malgenomeproject.org.

[3] Android marks fourth anniversary since launch with 75.0quarter, according to idc. https://www.idc.com/getdoc.jsp?containerId=prUS23771812.

[4] Android ndk. http://developer.android.com/tools/sdk/ndk/index.html.

[5] Android open source project. android security overview. http://source.android.com/tech/security.

[6] Angry birds. http://www.angrybirds.com.

[7] Linpack for android - android apps on google play. https://play.google.com/store/apps/details?id=com.greenecomputing.linpack&hl=en.

[8] Package java.lang.instrument. http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html.

[9] Phonegap. http://phonegap.com.

[10] The security manager. http://docs.oracle.com/javase/tutorial/essential/environment/security.html.

[11] Testing fundamentals. http://developer.android.com/tools/testing/testing_android.html.

[12] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard: enforcing user requirements on android apps. In *Proceedings of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2013), TACAS'13, Springer-Verlag, pp. 543–548.

[13] Bousquet, A., Briffaut, J., Clévy, L., Toinard, C., and Venelle, B. Mandatory Access Control for the Android Dalvik Virtual Machine. In *Workshop on Embedded Self-Organizing Systems (ESOS)* (2013).

[14] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., and Sadeghi, A.-R. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr. 2011.

[15] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards taming privilege-escalation attacks on android. In *Proceedings of 19th Annual Network & Distributed System Security Symposium* (Feb 2012), NDSS '12.

[16] Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 51–62.

[17] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Towards a framework for android security modules: Extending se android type enforcement to android middleware. Tech. Rep. TUD-CS-2012-0231, Center for Advanced Security Research Darmstadt, Nov. 2012.

[18] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proceedings of the 22nd USENIX conference on Security symposium* (2013), USENIX Security '13, USENIX.

[19] Conti, M., Nguyen, V. T. N., and Crispo, B. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th international conference on Information security* (Berlin, Heidelberg, 2011), ISC '10, Springer-Verlag, pp. 331–345.

[20] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In *Proceedings of the Workshop on Mobile Security Technologies* (San Francisco, CA, May 2012), MOST '12.

[21] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX conference on Security symposium* (Berkeley, CA, USA, 2011), USENIX Security '11, USENIX Association, pp. 23–23.

[22] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI '10, USENIX Association, pp. 1–6.

[23] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A study of android application security. In *Proceedings of the 20th USENIX conference on Security symposium* (Berkeley, CA, USA, 2011), USENIX Security '11, USENIX Association, pp. 21–21.

[24] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 235–245.

[25] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 627–638.

[26] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX conference on Security symposium* (Berkeley, CA, USA, 2011), USENIX Security '11, USENIX Association, pp. 22–22.

[27] Gong, L. Java security architecture (jdk1.2). http://docs.oracle.com/javase/1.4.2/docs/guide/security/spec/security-spec.doc.html, 1998.

[28] Hao, H., Singh, V., and Du, W. On the effectiveness of api-level access control using bytecode rewriting in android. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security* (2013), AsiaCCS '13.

[29] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: retrofitting android to protect data from

imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 639–652.

[30] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 3–14.

[31] Luo, T., Hao, H., Du, W., Wang, Y., and Yin, H. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, ACM, pp. 343–352.

[32] Nauman, M., Khan, S., and Zhang, X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2010), AsiaCCS '10, ACM, pp. 328–332.

[33] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference* (Washington, DC, USA, 2009), ACSAC '09, IEEE Computer Society, pp. 340–349.

[34] Pearce, P., Felt, A. P., Nunez, G., and Wagner, D. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (2012), AsiaCCS '12.

[35] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H. J., and Cowan, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (may 2012), SP '12, pp. 224 –238.

[36] Shekhar, S., Dietz, M., and Wallach, D. S. Adsplit: separating smartphone advertising from applications. In *Proceedings of the 21st USENIX conference on Security symposium* (Berkeley, CA, USA, 2012), USENIX Security '12, USENIX Association, pp. 28–28.

[37] Siefers, J., Tan, G., and Morrisett, G. Robusta: taming the native beast of the jvm. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 201–211.

[38] Smalley, S., and Craig, R. Security enhanced (se) android: Bringing flexible mac to android. In *Proceedings of 20th Annual Network & Distributed System Security Symposium* (Feb 2013), NDSS '13.

[39] Sun, M., and Tan, G. Jvm-portable sandboxing of java's native libraries. In *Proceedings of the 17th European Symposium on Research in Computer Security* (2012), vol. 7459 of *ESORICS '12*, pp. 842–858.

[40] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM symposium on Operating systems principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 203–216.

[41] Xu, R., Saïdi, H., and Anderson, R. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium* (Berkeley, CA, USA, 2012), USENIX Security '12, USENIX Association, pp. 27–27.

[42] Yan, L. K., and Yin, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium* (Berkeley, CA, USA, 2012), USENIX Security '12, USENIX Association, pp. 29–29.

[43] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2009), SP '09, IEEE Computer Society, pp. 79–93.

[44] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 95–109.

[45] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of 19th Annual Network & Distributed System Security Symposium* (Feb 2012), NDSS '12.