A Secure, Usable, and Transparent Middleware for Permission Managers on Android

Daibin Wang, Haixia Yao, Yingjiu Li, *Member, IEEE*, Hai Jin, *Senior Member, IEEE*, Deqing Zou, and Robert H. Deng

Abstract—Android's permission system offers an all-or-nothing choice when installing an app. To make it more flexible and fine-grained, users may choose a popular app tool, called *permission manager*, to selectively grant or revoke an app's permissions at runtime. A fundamental requirement for such permission manager is that the granted or revoked permissions should be enforced faithfully. However, we discover that none of existing permission managers meet this requirement due to *permission leaks*, in which an unprivileged app can exercise certain permissions which are revoked or not-granted through communicating with a privileged app. To address this problem, we propose a secure, usable, and transparent OS-level middleware for any permission manager to defend against the permission leaks. The middleware is provably secure in a sense that it can effectively block all possible permission leaks. The middleware is designed to have a minimal impact on the usability of running apps. In addition, the middleware is transparent to users and app developers and it requires minor modifications on permission managers and Android OS. Finally, our evaluation shows that the middleware incurs relatively low performance overhead and power consumption.

Index Terms—Permission manager, permission leaks, middleware, android

1 Introduction

At present, Android is the most widely used mobile OS [1] and it plays an important role in daily life [2]. To protect the sensitive resources on Android device, the permission model and the sandboxing mechanism are enforced on Android applications (*apps* for short). Unlike iOS [3], however, Android's permission model is an "all-or-nothing" approach. At install time, it provides users a binary choice of either accepting all the requested permissions or not installing an app. Once an app is installed, it is impossible to change the app's permissions unless a special app called *permission manager* is in use.

Permission managers can be used by users to selectively grant or revoke app's permissions at runtime. A variety of permission managers, such as LBE [4], Advanced Permission Manager [5], and XPrivacy [6], are provided in Google Play and some third-party Android markets. In addition, some popular custom ROMs or manufacturers' devices also include built-in permission managers, such as CyanogenMod [7], MIUI [8], and Huawei P6 [9]. Google also includes a hidden built-in permission manager, called App Ops, in Android from version 4.3[1] [10].

1. While App Ops is removed from Android 4.4.2, its framework still stays in the Android source code.

• *D. Wang, H. Jin, and D. Zou are with the Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {daibinwang, hjin, deqingzou}@hust.edu.cn.*
• *H. Yao, Y. Li, and R.H. Deng are with the School of Information Systems, Singapore Management University, Singapore.
E-mail: {hxyao, yjli, robertdeng}@smu.edu.sg.*

*Manuscript received 17 Mar. 2015; revised 31 Aug. 2015; accepted 10 Sept. 2015. Date of publication 17 Sept. 2015; date of current version 7 July 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2015.2479613*

A fundamental requirement for such permission managers is that the granted or revoked permissions should be enforced faithfully. However, we discover that *none* of the existing permission managers meet this requirement due to *permission leaks*. Permission leaks can be considered as an attack in which an unprivileged app can access certain sensitive resource without declaring the corresponding permissions [11], [12], [13], [14], [15], [16], [17], [18], [19].

Once vulnerable apps are installed in Android, the effectiveness of permission managers would be questionable. In this paper, we firstly explore the ability of existing permission managers to defend against permission leaks. After studying most of existing permission managers, including 14 real-world permission managers in four categories as detailed in Section 2.2.1, we discover a common and serious problem: *none of them could effectively defend against the permission leaks attack*, leaking permissions such as VIBRATE, CHANGE_WIFI_STATE, CALL_PHONE, SEND_SMS, and CAMERA, which is analyzed in Section 2.2.2. It means that an app could still access sensitive resources even though its user has revoked the corresponding permissions via certain permission manager. The result is serious because most users tend to trust permission managers in a sense that they could effectively enforce user-defined access control.

Motivated to address the above problem, we propose an OS-level middleware for existing permission managers to defend against permission leaks. The design goals of the middleware are: (i) effective defense against permission leaks, (ii) minimal impact on the usability of apps, (iii) transparent to app users and developers, (iv) minimal modification on permission manager and Android OS, and (v) low performance overhead. To manage a list of blocked permissions, we introduce the concept of *blocked permission list* (*BPL* for short). We use BPL instead of removing permissions from existing granted permission list since a blocked permission may not be chosen from the granted permission list. Also, the use of BPL reduces the impact on Android framework.

To detect the permission leaks at runtime and minimize the impact on the usability of apps, we propose a *multiple app instances based* approach (MAI). When an inter-component communication (*ICC* for short) occurs between two apps, MAI decides whether a new instance of the receiver app should be created based on the BPLs of the sender app and the receiver app. If a new instance is created, its BPL is the union of the sender app's and the receiver app's BPLs. Instances of single app share the same permission list while their BPLs may differ from each other. MAI effectively defeats permission leaks at runtime by enforcing permission control at app instance level. While the functionality of a newly created instance of certain app is restricted as necessary for defending against permission leaks, the other existing instances of the same app are not affected in our solution so that the usability of an Android device is not over restricted due to the upgraded security.

We implement a prototype of MAI-based middleware on Android Open Source Project [20] (AOSP, branch 4.4.2_r1). Analyses and experiments show that the middleware achieves its design goals. A brief summary of our contributions is given below.

- We take the first step to study the security problem of existing permission managers (Section 2.2). We discover that none of these permission managers are secure against permission leaks.
- We propose an OS-level middleware for existing permission managers to defend against permission leaks. A multiple app instance-based approach is used to implement the middleware (Sections 4 and 5). It can be proven that the middleware is effective in defending permission leaks (Section 6.1); it also minimizes the impact on the usability of apps (Section 6.2).
- Our middleware is transparent to app users and developers. It requires minor modifications to the permission managers and does not require any modification of existing apps (Section 6.3).
- Experiment results show that our middleware incurs low performance overhead and power consumption (Section 7).

The rest of this paper is organized as follows. Section 2 introduces the background knowledge and the motivation of this paper. Section 3.2 clarifies the threat model and design goals of our middleware. Section 4 presents the system design of the middleware. Section 5 gives implementation details. Section 6 analyzes the middleware from security, usability, and deployment perspectives. Section 7 shows the performance evaluation results. Section 8 summarizes the related work. Finally, Section 9 concludes the paper.

2 Background and Motivation

In this section, we provide the background knowledge on Android and permission leaks. We also investigate the issues of existing permission managers that motivate our research.

2.1 Android and Permission Leaks

App Components and ICC. Each Android app [21] is composed of four types of component: *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. An Activity component provides the user interface for an app. A Service component performs long-running operations in the background. The Broadcast Receiver component responds to system or application broadcast announcements. A Content Provider component manages a set of app data, which can be private or shared with other apps.

Android provides an inter-component communication (ICC) mechanism for components to make use of other component's functions. An ICC can be intra- or inter-application communication. In Android framework, *Intents* are the primary vehicle for ICC. Intent is an abstract description of an operation to be performed. For example, an app creates an Intent containing a URL and sends it to a Browser app that opens the specified web page. Intent is used with some ICC-related APIs to start or interact with components [22]. For example, `startActivity()` API is used to start an Activity component. To mediate the whole system's ICCs, *ActivityManagerService* (*AMS* for short) is created in Android framework. AMS is a system service run in a system process (named `system_server`), which is isolated from app processes.

Android security model. Android is designed with two basic security mechanisms: *application sandbox* and *permission model* [23]. As Android is built upon Linux kernel, it takes advantage of Linux user-based protection to enforce application sandbox. Android system assigns a unique user ID (UID) to each app and runs it in a separate process. To access sensitive resources on the device, Android provides *sensitive APIs*, which are also called *privilege APIs* or *protected APIs*, for any apps that have been granted with certain permissions. For example, `sendTextMessage()` API is used to send a text via SMS and it requires that an app should have been granted the SEND_SMS permission. To make use of these protected APIs, an app should declare a *permission list* in its manifest file. At install time, Android provides for the user a binary choice of either accepting all the requested permissions or not installing the app. The user can not grant or deny individual permissions. Once granted, the permissions are applied to the app as long as it is installed. At runtime, Android would not notify user again when an app accesses sensitive resources.

By default, all components of the same app run in a single process, which is also called *app instance* in the paper. Nonetheless, app developers may choose to make different components of the same app run in separate processes. The permission lists of such separate processes are identical to the permission list that is claimed in the app's manifest file. In AMS, the `ProcessRecord` class is used to show the information, e.g., process name, PID, and etc., of a running process.

Permission leaks. Permission leaks happen when an unprivileged app accesses sensitive resources through ICCs with a vulnerable app which is granted with the required permissions. For example, app1 without permission p could not access any sensitive resource protected by permission p, but app1 may request app2 with permission p to access some sensitive resources. In this example, app2 is a *vulnerable app* which consists of some vulnerable components that can be

TABLE 1
Five Known Cases of Permission Leaks

Case ID	Package Name	Exposed Component	Leaked Capability
1	com.android.phone [11]	PhoneGlobals$PhoneAppBroadcastReceiver	VIBRATE
2	com.android.settings [11]	widget.SettingsAppWidgetProvider	CHANGE_WIFI_STATE
3	com.android.phone [17]	PhoneGlobals$NotificationBroadcastReceiver	CALL_PHONE
4	com.android.mms [19]	transaction.SmsReceiverService	SEND_SMS
5	com.android.camera2	CameraActivity	CAMERA

potentially exploited by other apps. In general, vulnerable app can be system app [11], [12], [14], [15], [16], [17], [19] and third-party apps [13], [18]. Permission leaks are also called capability leaks [12], [14], [15] and permission re-delegation [11] in the literature.

Permission leaks are caused by accidentally or intentionally exposing an app's internal functionality to other apps. Since the Android permission model only checks the permissions of a requesting app, it opens a door for any app to exploit the permission leaks and bypass the permission model.

The prevalence of permission leaks on system apps and third-party apps has been exposed recently [12], [13], [24], [25]. For instance, IntentFuzzer [24] detects that 161 of 2,183 top free apps in Google Play have at least one permission leak. However, it is not an easy work to discover and fix these vulnerable apps through static analysis and fuzzing technology. For example, it takes more than 3 years to discover and fix the the CVE-2013-6272 vulnerability [17].

2.2 Permission Managers and Its Issues

Since much personal data are stored on smartphones, one of the most pressing issues of smartphone security is to protect user privacy. Since Android's permission model offers an "all-or-nothing" choice when installing an app, a user cannot selectively grant or revoke the app's permissions at install time or runtime. Therefore, what the user hopes is that the developer of the app is benign and conforms to the Principle of Least Privilege (PLP) in designing the app. Unfortunately, various studies have shown that most of Android apps abuse their permissions [18], [33], [34], [35].

To make the Android's permission model more flexible and fine-grained, users may choose a popular tool, called *permission manager*,[2] to selectively grant or revoke an app's declared permissions at runtime. However, there exist some common issues with the existing permission managers.

2.2.1 Existing Permission Managers

In general, existing permission managers in Android can be divided into four categories, including APK modification, root based, App Ops based, and ROM based permission managers.

APK Modification. APK modification is to rewrite an app's declared permission list on its manifest file and generate a new APK file accordingly. The new APK is then installed to replace the old version. Advanced Permission Manager [5] is one example of such permission managers.

Root based. Some apps enforce permission management through process injection. Once a shared library object file is injected into a target process, it hooks the privilege APIs and checks against user-defined policies. Due to the sandbox protection in Android, the root based approach requires that the Android device be rooted.

App Ops based. In Android 4.3 and 4.4, Google provides a hidden permission manager called App Ops. Many third-party apps (e.g., App Ops Starter [31]) provide the function of permission management through invoking the interface of App Ops. Since App Ops is built into the Android ROM, those apps require no modification on Android framework or rooted device.

ROM based. Apart from permission managers as third-party apps, several ROMs (e.g., CyanogenMod [7], MIUI [8], Huawei [9]) are equipped with the functions to manage apps' permissions. Because these permission managers are implemented as a part of ROM, there is no need for users to root their devices.

2.2.2 Issues with Existing Permission Managers

Whatever technique is used by a permission manager, a fundamental requirement is that it should effectively prevent apps from bypassing user-defined access control through permission leaks. To examine the effectiveness of existing permission managers, we collect five known cases of permission leaks and fourteen typical permission managers in our investigation. Table 1 shows the exposed components and leaked capabilities of these cases.[3] The name, category, and version of the tested permission managers are shown in Table 2.

Five attacker apps are built to exploit the exposed component of the corresponding vulnerable app through ICC. Each attacker app does not declare the corresponding leaked permission in its manifest file while it attempts to access the sensitive resources protected by the leaked permission. We install an attacker app, a vulnerable app (if required), and a permission manager in a single device. The permission manager is used to block the leaked permission for the attacker app. Then, we observe whether the attacker app can still access the sensitive resources protected by the blocked permission through the vulnerable app. We repeat the above procedure for fourteen permission managers.

If an attacker app does not declare any leaked permission in its manifest file, unfortunately, we discover that none of the 14 permission managers allows users to block any permission beyond the claimed permissions. Nonetheless, an attacker app can exploit permission leaks and access

2. The tools managing the root privilege, such as SuperSU [36], are out of the scope of this paper.

3. While Android allows third-party apps taking photos through the system camera app without declaring the CAMERA permission, we still consider sample 5 as a case of permission leaks.

TABLE 2
Fourteen Tested Permission Managers and Their Results (✓: Successfully Prevent, N/A: Not Support)

ID	Name	Category	Version	Case				
				1	2	3	4	5
1	SRT AppGuard [26]	APK modification	2.2.7	N/A				
2	Advanced Permission Manager [5]	APK modification	3.2.2					
3	Permission Manager [27]	APK modification	1.1					
4	LBE Security Guard [4]	Root based	5.4.7896	N/A				✓
5	XPrivacy [6]	Root based	3.4.13					✓
6	App Settings [28]	Root based	1.1					
7	DonkeyGuard [29]	Root based	0.5.67	N/A				
8	Tencent Mobile Manager [30]	Root based	5.1.0	N/A				
9	App Ops Starter [31]	App Ops based	1.1.0	✓	✓			
10	Permission Manager [32]	App Ops based	2.0	✓	✓			
11	App Ops on Android 4.3 [10]	ROM based	–	✓	✓			
12	Privacy Guard on CyanogenMod 10.2.0 [7]	ROM based	–			✓		
13	Permission Manager on MIUI v5 [8]	ROM based	4.0.1114	N/A				✓
14	Permission Manager on Huawei P6 [9]	ROM based	6.1.9	N/A				

sensitive information without having the corresponding permissions in its claimed permission list. This shows the incompleteness of the permission controls of the existing permission managers.

Next, consider the case in which an attacker app declares the leaked permission in its manifest file. The testing results are shown in Table 2. Note that since some permission managers only allow users to configure privacy-related permissions, case 1 is not applicable to them. Table 2 shows that while some of the tested permission managers can block one or two cases of tested permission leaks, none can effectively block all cases. The results show that the existing permission managers are vulnerable to the permission leaks. Once a vulnerable app is installed on a device, other apps could exploit it and bypass the access control of permission managers. Consequently, such attacks break the trust of users to the permission managers.

A ideal solution to solve above problems is to locate and fix all vulnerable components. At present, however, there is no effective solution that could locate every vulnerable component. Once a vulnerable app, especially system app, is installed, it could be difficult to fix its vulnerable components.

3 Threat Model and Design Goals

In this paper, we propose an OS-level middleware for Android permission managers to effectively defend against permission leaks at runtime. The threat model and design goals of the middleware are presented in this section.

3.1 Threat Model

It is assumed that users download and install third-party apps. These third-party apps, as well as system apps, may have vulnerable components that can be exploited to incur permission leaks. However, these apps do not collude in sharing information. In addition, it is assumed that Android OS, including Android kernel and Android framework, are trusted.

3.2 Design Goals

The design of the middleware is guided by following goals:

- *Effective prevention.* The middleware should be able to effectively defend against permission leaks at runtime. Any Android permission manager supported by our middleware can thus faithfully enforce its granted or revoked permissions in the presence of vulnerable apps.
- *Minimal impact on usability.* While the functionality of apps is affected as necessary for defending against permission leaks, our middleware should minimize the impact on the usability of any running app.
- *Compatible.* The middleware is transparent to app users and developers. It provides public interfaces for existing permission managers to tap on so as to be secure against the permission leaks.
- *Minimal modification.* The design and implementation of our middleware require minor changes to existing permission managers and Android OS.
- *Low performance overhead.* The middleware incurs reasonably low performance overhead which does not affect user's experience significantly.

We will show that our middleware meets the design goals. In particular, the effective prevention goal is achieved by attaching a list of blocked permissions to each app instance which has precedence over the list of permissions granted to the app instance. When an app instance is called by another app instance, the blocked permission list of the callee instance grows to be the union set of the blocked permission lists of both caller instance and callee instance. The goal of minimal impact on usability is achieved by checking blocked permissions at the app instance level, instead of app level. The compatible goal is achieved by providing public interfaces for any permission managers to call and by requesting no changes to app development or user experience. The minimal modification goal is achieved by making minimum changes to the Android framework and permission managers in implementation. Finally, the goal of low performance overhead is achieved by optimizing the design and implementation of our middleware, which is detailed in the following sections.

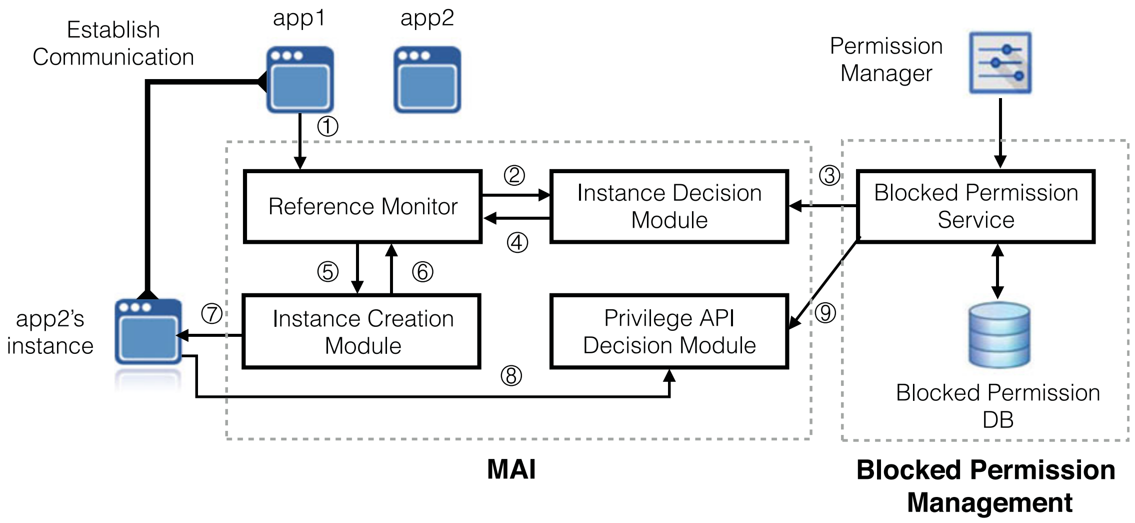

Fig. 1. Workflow of our middleware.

4 System Design

In this section, we present the design of our middleware, including blocked permission list, workflow, instance decision, and privilege API decision.

4.1 Blocked Permission List (BPL)

In Android, each installed app has a *permission list* which represents a set of permissions granted by user. To distinguish from such permission list, we define another permission list named *blocked permission list* for each app, which represents a set of permissions blocked by user. Instead of removing permissions from the existing permission list, BPL is suitable for permission manager to grant or revoke any permissions, even beyond those defined in the existing permission list, at runtime. Our design enables a user to block any permission that is not declared in the app's manifest file but in the BPL that is provided by a permission manager app. Our middleware enforces the access control based on both BPLs and declared permission lists.

4.2 Workflow

The workflow of our middleware is shown in Fig. 1. It consists of two major blocks: (i) Blocked Permission Management; and (ii) MAI.

Blocked permission management. The blocked permission management interacts with permission managers and is responsible for managing the BPLs provided by permission managers. There are two sub-blocks within the blocked permission management, including Blocked Permission DB and Blocked Permission Service.

The *Blocked Permission DB* is a database to store the BPLs provided by permission managers. The database should be updated when an app's BPL is changed or when an app is uninstalled.

The *Blocked Permission Service* is a system service which is responsible for managing BPLs in the Blocked Permission DB. The Blocked Permission Service provides the interfaces for other system services and permission managers. To prevent unauthorized entities from exploiting these public interfaces, only trusted permission managers and system services are allowed to communicate with the service.

MAI. The Multiple App Instances based approach is designed for defending against the permission leaks at runtime. There are four sub-blocks within MAI, including Reference Monitor, Instance Decision Module, Instance Creation Module, and Privilege API Decision Module.

The *Reference Monitor* is responsible for intercepting ICC communications. For example, when app1 sends an ICC communication request to app2, this request is intercepted by the Reference Monitor (*step* ① in Fig. 1). The Reference Monitor then transmits the request to the Instance Decision Module (*step* ② in Fig. 1). If a new instance is required, the Reference Monitor makes a request to the Instance Creation Module (*step* ⑤ in Fig. 1). Note that when an instance is created, its BPL is stored in the Blocked Permission DB through the Blocked Permission Service.

The *Instance Decision Module* is responsible for deciding whether a new instance should be created based on BPLs of caller and callee app. At first, the caller's and callee's BPLs are retrieved through the Blocked Permission Service (*step* ③ in Fig. 1). Then, it makes a decision based on the BPLs and returns the decision result back to the Reference Monitor (*step* ④ in Fig. 1). A detailed algorithm of the module is given in Section 4.3.

The *Instance Creation Module* is responsible for creating a new instance and returning the PID of it back to the Reference Monitor (*step* ⑥ in Fig. 1). For example, an instance of app2 is created and we establish a communication between app1 and the instance (*step* ⑦ in Fig. 1). Note that the new instance's privilege may be restricted based on the result of Instance Decision Module.

When an instance or an app invokes a privilege API, the privilege API request is intercepted by the *Privilege API Decision Module* (*step* ⑧ in Fig. 1). The Privilege API Decision Module determines whether the app or instance is allowed to access the sensitive resources (*step* ⑨ in Fig. 1). A detailed algorithm of this module is given in Section 4.4.

4.3 Instance Decision

In MAI, each app may have several instances at runtime which share the same permission list. However, each instance's BPL is different from each other. Note that a permission list is a whitelist of granted permissions, while a BPL is a blacklist of blocked permissions. An instance decision in MAI occurs when an ICC is invoked and the decision is based on the caller and callee apps' BPLs. The instance decision flow is formulated in Algorithm 1. First of all, the BPLs of the caller and callee apps are retrieved from the Blocked Permission DB (Algorithm 1, Line 1-2). If the caller's BPL is a subset of the callee app's BPL, it notifies the Reference Monitor that no new instance needs to create (Algorithm 1, Line 3-5). Otherwise, the union set of the caller and callee BPLs is calculated and the whole running instances of the callee app are retrieved (Algorithm 1, Line 6-7). To reduce the number of instances, if any instance of the callee app has the same BPL as the union set, it notifies the Reference Monitor that no new instance needs to create and the information (e.g., PID) of the target instance is also provided (Algorithm 1, Line 8-12). Otherwise, it notifies the Reference Monitor to create a new instance and the BPL of new instance is the union set (Algorithm 1, Line 13). Thus, the privilege of the new instance is more restricted because its BPL is the union set of the BPL of the caller app and the BPL of the callee app.

Algorithm 1. Instance Decision Procedure

```
Input:
    Caller app $caller$ and Callee app $callee$;
Output:
    Decision result $isCreate$ (and new instance's BPL $BPL_{new}$);
 1: retrieve the BPL $BPL_{caller}$ of caller
 2: retrieve the BPL $BPL_{callee}$ of callee
 3: if $BPL_{caller} \subset BPL_{callee}$ then
 4:     return $isCreate = false$
 5: end if
 6: $BPL_{union} = BPL_{caller} \cup BPL_{callee}$
 7: retrieve all runtime instances $Ins$ of callee app
 8: for all $Ins_i \in Ins$ do
 9:     retrieve the BPL $BPL_{Ins_i}$ of instance $Ins_i$
10:     if $BPL_{Ins_i} = BPL_{union}$ then
11:         return $isCreate = false$ and the target instance ID
12:     end if
13:     return $isCreate = true$ and $BPL_{new} = BPL_{union}$
14: end for
```

4.4 Privilege API Decision

In MAI, the privilege API decision is made based on the caller's permission list and BPL. We present the procedure of privilege API decision in Algorithm 2. At first, the BPL of the caller instance is retrieved from the Blocked Permission DB through the Blocked Permission Service (Algorithm 2, Line 1). If any required permission is in the caller's BPL, it denies the privilege API request because a BPL has precedence over the corresponding permission list (Algorithm 2, Line 2-6). Then, it retrieves the app's permission list of the caller app from Android framework (Algorithm 2, Line 7). If the permission list contains the required permission list of privilege API, it grants the request (Algorithm 2, Line 8-10). Otherwise, it denies the request (Algorithm 2, Line 11).

5 Implementation

Our middleware is implemented within the Android framework.

Algorithm 2. Privilege API Decision Procedure

```
Input:
    Caller instance $caller$ and required permission list $Perm_{required}$
Output:
    Decision result $isAllow$;
 1: retrieve the BPL $BPL_{caller}$ of caller instance
 2: for all $Perm_i \in Perm_{required}$ do
 3:     if $Perm_i \in BPL_{caller}$ then
 4:         return $isAllow = false$
 5:     end if
 6: end for
 7: retrieve the permission list $Perm_{caller}$ of caller instance
 8: if $Perm_{caller} \supset Perm_{required}$ then
 9:     return $isAllow = true$
10: end if
11: return $isAllow = false$
```

5.1 Blocked Permission Database

A blocked permission database is created in the Android framework and the database is protected by standard Unix-like access permissions. It includes two tables: *appBPL* and *instanceBPL*. The BPL of each app provided by a permission manager is stored in the appBPL table, while the instanceBPL table stores the BPLs of created instances. The appBPL table is indexed by app's UID, while the instanceBPL is indexed by instance's PID and UID. When the system shutdowns, the instanceBPL table is cleared because all created instances are destroyed. When an app is uninstalled, its BPLs on the appBPL and instanceBPL tables are also cleared.

5.2 Blocked Permission Service

The Blocked Permission Service is implemented as a new system service run in the `system_server` process and it manages the blocked permission database through the SQLite interface provided by Android. To be compatible for existing permission managers, the service provides `addBPL()` and `modifyBPL()` interfaces for permission manager apps to manage the BPLs of apps. In addition, it provides `retrieveBPL()` API for system services to retrieve an app's BPL.

The public interfaces provided by the Blocked Permission Service are designed to allow only trusted permission managers and system services to communicate with them. To defend against unprivileged apps from accessing these interfaces, the Blocked Permission Service checks the identity of a caller in *two ways*. A *uid checking* is used to decide whether the caller is a system service. Since our current prototype requires AMS to retrieve BPLs from the Blocked Permission Service, the `system_server` process, whose uid is 1,000, is allowed to invoke the interfaces. If the caller is an app, *certificate and signature verification* is used to verify the identity of the caller. The trust between permission manager apps and Blocked Permission Service can be built via the Public Key Infrastructure (PKI). Any permission manager that requires to communicate with our middleware should firstly be issued a certificate signed by a trusted certificate authority (CA). A parameter of the public interface is used to deliver the certificate of a permission manager to the Blocked Permission Service, which verifies the certificate through certain standard Java APIs. Trusted CA's certificates should be stored in the ROM at build time. X.509 format certificates are supported in the current prototype.

5.3 Instance Decision Module and Instance Creation Module

Both Instance Decision Module and Instance Creation Module are implemented in the AMS. The Reference Monitor provided by AMS is instrumented to notify the Instance Decision Module when an ICC occurs. The Instance Decision Module implements the instance decision procedure with the help of `retrieveBPL()` API. The Instance Creation Module uses a system API, i.e., `startProcessLocked()`, to create a new instance and it stores the instance's BPL in the instanceBPL table.

To distinguish different instances of the same app, two variables: `processAliasName` and `processBPL`, are added to the `ProcessRecord` class in the AMS. The `processAliasName` is used to distinguish the instances created by MAI from the instances created by Android; this variable is assigned with the process name of caller and callee instances. On the other hand, the `processBPL` consists of the BPL of each instance. The purpose of storing BPL in memory for each instance is to improve the performance of our middleware. To reduce the cost of memory, the `processBPL` variable is represented as an array of integers and each integer maps to a permission in Android. Since the number of permissions on Android is relatively small (e.g., 145 permissions in Android 4.4.2), the memory overhead of our middleware is negligible. For example, if there are 1,000 app instances on an Android 4.4.2 device , no more than 1 MB memory is used to store the BPLs of these app instances.

5.4 Privilege API Decision Module

The Privilege API Decision Module is also implemented in the AMS.[4] A hook is placed for checking the blocked permissions in the regular Android permission checking. The public method `checkPermission(permission, pid, uid)` inside AMS is the only public entry point for all permission checking in Android framework. The three parameters of the method represent the required permission, the PID of the caller instance, and the UID of the caller instance, respectively. The BPL of caller is retrieved through `retrieveBPL()` API. Before the Android permission checking is performed inside `checkPermission` method, the logic for checking the blocked permissions is added and enforced.

6 Analysis on Our Middleware

In this section, we analyze how our middleware achieves the first four goals presented in Section 3.2. We firstly analyze the security of our middleware under the threat model, which is given in Section 3.1. Then, we analyze the usability of our middleware as compared with other possible solutions. Finally, we discuss how to deploy our middleware in practice.

4. Android's permission system is enforced in two levels: framework and kernel. For the current prototype system, we just focus on the permissions enforced in the Android framework and leave the kernel-enforcing permissions as our future work.

6.1 Security

We assume that an Android device has been flashed into a custom ROM with our middleware integrated. A vulnerable app $\mathcal{A}$ that can be exploited to leak permission $\mathcal{P}$ is installed in the device. Thus, the permission list of app $\mathcal{A}$ contains permission $\mathcal{P}$ ($\mathcal{P} \in Perm_{\mathcal{A}}$). Consider another app $\mathcal{B}$ installed in the same device for which the user blocks the permission $\mathcal{P}$ through a permission manager supported by our middleware; that is, the BPL of app $\mathcal{B}$ contains permission $\mathcal{P}$ ($\mathcal{P} \in BPL_{\mathcal{B}}$). We will prove that the permission manager could effectively block the permission $\mathcal{P}$ of app $\mathcal{B}$.

Under the assumption that the Android framework is trusted, privilege request would be intercepted by our middleware. Since the BPL of app $\mathcal{B}$ includes the permission $\mathcal{P}$, app $\mathcal{B}$ cannot directly access the sensitive resource protected by permission $\mathcal{P}$ no matter whether or not the permission list of app $\mathcal{B}$ includes the permission $\mathcal{P}$.

There are two possible ways for app $\mathcal{B}$ to bypass the access control of the permission manager. One is to changes its BPL if app $\mathcal{B}$ has the permission $\mathcal{P}$. In our implementation, the blocked permission database can only be accessed by the Blocked Permission Service, i.e., `system_server` process. Other processes cannot modify the database due to the process isolation of Android OS. The certificate and signature verification performed in our middleware prevents app $\mathcal{B}$ from changing its BPL through the public interfaces provided by the Blocked Permission Service.

Another possible way for app $\mathcal{B}$ to bypass the access control of the permission manager is to exploit app $\mathcal{A}$. Because Android framework and Linux kernel are trusted, the only communication channel between app $\mathcal{A}$ and app $\mathcal{B}$ is ICC. When an ICC is invoked by app $\mathcal{B}$, our middleware decides whether a new instance should be created. The instance's BPL is the union of app $\mathcal{A}$ and app $\mathcal{B}$'s BPLs ($BPL_{Ins} = BPL_{\mathcal{A}} \cup BPL_{\mathcal{B}}$). Since permission $\mathcal{P}$ is in the BPL of app $\mathcal{B}$, permission $\mathcal{P}$ is also in the instance's BPL ($\mathcal{P} \in BPL_{Ins}$). Therefore, the new instance also cannot access the sensitive resource protected by permission $\mathcal{P}$.

A use case scenario. Alice has an Android 4.4.2 smartphone, into which our middleware has been flashed. Due to the inflexibility of Android permission system, she installs a permission manager, which is trusted by our middleware, to control the permissions of apps. Assuming that Alice likes to listen music using the device, and she installs a music app from Google Play on her device. Unfortunately, the music app is a malicious app and it can exploit the SMS resend vulnerability [19] on Android 4.4.2. The SMS resend vulnerability is due to a vulnerable component, named *SmsReceiverService*, of the system sms app, and it allows any app to send SMS messages without the SEND_SMS permission.

As a security-sensitive user, Alice blocks the SEND_SMS permission on the music app through the trusted permission manager because is is unnecessary for a music app to send SMS. To defend against permission leaks, the trusted permission manager passes the BPL of the music app, i.e., SEND_SMS permission, to our middleware.

Since the SEND_SMS permission is included in the BPL of the malicious music app, the app could not send any SMS directly. Therefore, the app tries to indirectly send a SMS through the vulnerable component of the system sms app. When it sends an ICC request to the sms app, an instance of the sms app is created to serve the request. The instance inherits the BPLs of music app and sms app; in other words, the instance's BPL contains the SEND_SMS permission. Thus, the instance also can not send any SMS. This shows that the music app cannot send any SMS after its SEND_SMS permission is blocked by the permission manager.

6.2 Usability

There exist other techniques such as the call-chain based method [37] and the graph-based method [38], which can be potentially used to defend against permission leaks for permission managers. One common feature of these methods is that the permission checking of any app is based on its UID; in other words, all instances of an app are restricted the same way for defending against permission leaks. For simplicity, we name the above methods as *simple design*. In this section, we demonstrate that the usability of our design is better than the simple design.

We define the *usability* of an app $\mathcal{A}$ as the available permission set $AP_{\mathcal{A}}$ of app $\mathcal{A}$. The larger the available permission set is, the better the usability. Assuming that app $\mathcal{A}$ has n instances (i.e., $\mathcal{A} = \{Ins_1, Ins_2, \dots, Ins_n\}$), given its permission list $Perm_{\mathcal{A}}$ and blocked permission list $BPL_{\mathcal{A}}$. The available permission set of app $\mathcal{A}$ includes the available permission set of each instance in app $\mathcal{A}$. For any instance Ins_i ($i \in [1, n]$) of app $\mathcal{A}$, its initial available permission set is given below

$$AP_{\mathcal{A}}^{Ins_i} = Perm_{\mathcal{A}} \setminus (Perm_{\mathcal{A}} \cap BPL_{\mathcal{A}}).$$

Next, we assume that app $\mathcal{A}$ is invoked by app $\mathcal{B}$ directly. A new instance Ins_m ($m \notin [1, n]$) of app $\mathcal{A}$ is created in our design while no instance is created in the simple design. Thus, in the simple design, the available permission set of an instance Ins_i ($i \in [1, n]$) in app $\mathcal{A}$ is

$$AP_{\mathcal{A}}^{Ins_i} = Perm_{\mathcal{A}} \setminus (Perm_{\mathcal{A}} \cap (BPL_{\mathcal{A}} \cup BPL_{\mathcal{B}})).$$

In our design, the available permission set of an instance Ins_i in app $\mathcal{A}$ is

$$AP_{\mathcal{A}}^{Ins_i} = \begin{cases} Perm_{\mathcal{A}} \setminus (Perm_{\mathcal{A}} \cap (BPL_{\mathcal{A}} \cup BPL_{\mathcal{B}})) & i = m \\ Perm_{\mathcal{A}} \setminus (Perm_{\mathcal{A}} \cap BPL_{\mathcal{A}}) & i \in [1, n]. \end{cases}$$

For the instances Ins_i ($i \in [1, n]$) of app $\mathcal{A}$, the available permission sets in the simple design are smaller than the initial available permission set. However, the available permission sets of them in our design are the same as the initial available permission set. In this sense, the usability of our design is better than the simple design.

Further, we assume that there is a call chain from app $\mathcal{B}$ to app $\mathcal{A}$ through some other apps (i.e., $CallChain = \{\mathcal{B}, D_1, \dots, D_p, \mathcal{A}\}$) and a new instance Ins_m ($m \notin [1, n]$) of app $\mathcal{A}$ is created to serve the request in our design. Thus, in the simple design, the available permission set of an instance Ins_i ($i \in [1, n]$) in app $\mathcal{A}$ is

$$AP_{\mathcal{A}}^{Ins_i} = Perm_{\mathcal{A}} \setminus \left(Perm_{\mathcal{A}} \cap \left(\bigcup_{j=1}^{p} BPL_{D_j} \cup BPL_{\mathcal{A}} \cup BPL_{\mathcal{B}} \right) \right).$$

In our design, the available permission set of an instance Ins_i in app $\mathcal{A}$ is

$$AP_{\mathcal{A}}^{Ins_i} = \begin{cases} Perm_{\mathcal{A}} \setminus (Perm_{\mathcal{A}} \cap (\bigcup_{j=1}^{p} BPL_{D_j} \cup BPL_{\mathcal{A}} \cup BPL_{\mathcal{B}})) & i = m \\ Perm_{\mathcal{A}} \setminus (Perm_{\mathcal{A}} \cap BPL_{\mathcal{A}}) & i \in [1, n]. \end{cases}$$

Clearly, for the instances Ins_i ($i \in [1, n]$) of app $\mathcal{A}$, the available permission sets of them in our design is larger than the simple design.

Therefore, we conclude that the usability of our design is better than the simple design. While achieving better usability, our design may incur certain performance overhead due to the creation of new app instances. In Section 7, we evaluate the performance of our middleware and show that the overhead is acceptable in practice.

6.3 Deployment

In this section, we discuss the deployment of our designed middleware in the following aspects: *Android framework*, *third-party apps*, *permission managers*, and *Public Key Infrastructure*.

Android framework. Our middleware requires to deploy a custom ROM on devices. About 1,200 lines of source codes on Android framework should be added or modified for our middleware. Since it requires to deploy a custom ROM, the adoption of our middleware would suffer from the same drawbacks of any ROM based approaches. Nevertheless, most parts of these source codes are new modules or services decoupled from the Android's core modules. Therefore, it is relatively easy for manufacturers and Google to integrate our middleware.

Third-party apps. Because our middleware does not change any existing API provided for apps, it is transparent to third-party app developers. Therefore, no modification on third-party apps is required.

Permission managers. Permission managers should be modified so that they can make use of our middleware. Because our middleware provides public interfaces, minor changes are required for permission managers. For example, only about 20 lines of source codes in App Ops are modified for using our middleware. It is easy to modify permission managers to allow to block any permission of apps even they are not in the claimed permission lists. Developers of permission managers are also required to register PKI certificates in order to communicate with our middleware.

Public Key Infrastructure. To prevent unauthorized entities from exploiting the public interfaces provided for permission managers, we suggest that any incoming BPL to the middleware should come with a signature and a certificate of permission manager. The certificates of permission managers should be issued by trusted certification authorities (CAs) in the PKI infrastructure. The CA's certificate should be stored in the ROM at build time.

7 Performance Evaluation

In this section, we evaluate the performance overhead of our middleware on Android device and show how our middleware achieves the last goal presented in Section 3.2. A prototype system of our middleware is implemented on Android 4.4.2_r1 (KitKat) and it is flashed into a Google Nexus 5 phone (Qualcomm Snapdragon$^{\text{TM}}$ 800 2.26 GHz CPU, Adreno 330

TABLE 3
AnTuTu Benchmark Results

AnTuTu	Baseline	Our System	Overhead
Total Score	37,510.10	36,595.73	2.44%
RAM	1,521.00	1,419.91	6.65%
CPU int	1,939.50	1,933.36	0.32%
CPU fp	2,323.90	2,317.91	0.26%
2D	1,639.10	1,638.91	0.01%
3D	12,483.20	12,275.55	1.66%
Database	650.50	646.36	0.64%

450 MHz GPU, 2 GB RAM, 16 GB Internal Storage, 2,300 mAh battery). All experiments are conducted on the same phone. App Ops is used as the tested permission manager.

7.1 Overall Performance

We use a comprehensive benchmark tool, AnTuTu 5.1 [39], to evaluate the overall performance of our system. AnTuTu is a popular tool that has more than 10 million downloads in the Google Play [40]. The benchmark tool runs a series of performance tests, including memory access, CPU integer & float point operation, graphics, SD card performance, and database I/O. The higher the score is, the better. We execute the benchmark 10 times and average the scores, which are shown in Table 3. The "baseline" means that the scores obtained from the device with stock Android on which our middleware is not installed.

From the table, we can observe that our system imposes a small impact on all but the memory operation (6.65 percent). Our system imposes the memory overhead because new app instances are created in memory in order to defend against permission leaks.

7.2 ICC and Permission Checking Overhead

Time overhead is introduced by our middleware when an ICC API or privilege API is invoked.

ICC overhead. When an app is run on the device with our middleware, the ICC procedure involves the process of checking the sender and receiver apps' BPLs and starting a new app instance if needed. We measure a single ICC overhead introduced by our middleware in three different settings: (1) "Stock", which means the results obtained from the stock Android without our middleware; (2) "MAI-no", which means that the results are obtained from a device with our middleware installed but no instance is created when an ICC is invoked; (3) "MAI-instance", which means that the results are obtained from a device with our middleware and a new instance is created when an ICC is invoked. Four ICCs (i.e., *startActivity()*, *startService()*, *bindService()*, and *sendBroadcast()*) are tested 50 times in each of the three settings.

We expect that our system incurs certain time overhead when a new instance is created. This is confirmed by our results in Fig. 2. If no instance is created, our system imposes no significant overhead (8.39 percent on average). If a new instance is created, our system's ICC procedure takes on average sixfold as long as Android's own ICC procedure. Most part of the time overhead is caused by spawning a new process. While the time overhead of our middleware is comparatively high, the overall time taken is

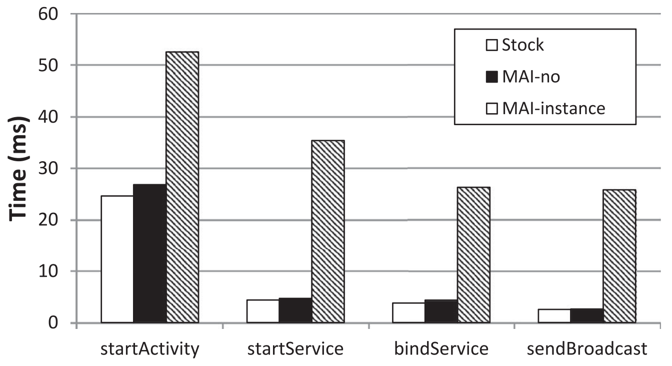

Fig. 2. Execution time of four ICCs under three settings.

less than 0.1 seconds; therefore, the user's experience is not significantly affected.

The ICC overhead of an app introduced by our system depends greatly on a variety of factors, including caller and callee apps' BPLs, the frequency of ICC, and total RAM of device. In the best case, i.e., no extra instance is created, the overhead is close to the overhead in the "MAI-no" setting. In the worst case, each ICC to the app would create a new instance. However, the worst case rarely happens in practice due to the following reasons: 1) Our middleware is designed to reduce the number of processes at runtime. For example, the instances with the same BPL are hosted in the same process of an app. 2) By counting the number of ICCs on the device, we discover that inter-app ICCs take only about 18.5 percent of total ICCs while the rest of ICCs do not generate any new instance.

Permission checking overhead. We measure the permission checking overhead in the following settings: (1) "Stock", which means the results obtained from the stock Android without our middleware; (2) "BPL-no", which means that the results are obtained from a device with our middleware and that the checked permissions are not in the corresponding BPL; (3) "BPL", which means that the results are obtained from a device with our middleware and that the checked permissions are in the corresponding BPL. Eight permissions are randomly selected and tested 50 times in each of there settings.

Fig. 3 shows that the time overhead in the "BPL-no" setting is negligible (about 5.45 μs increase per call). From Table 3, we note that the time decreases in the "BPL" setting. This is due to the logic of our permission checking: once the checked permission is in the app's BPL, our middleware will not check the app's permission list any more.

7.3 Energy Consumption and Memory Usage

To measure the energy and memory usage introduced by our middleware, we conduct the following tests. Every 10 minutes we sequentially run three apps: Messaging, WordPress, and Facebook, on a full-charged Nexus 5 phone via a monkeyrunner [41] script. All apps perform the following operations: taking a picture and sending a message. The experiment lasts for a period of 120 minutes. We execute this experiment under three environments: stock Android, our middleware with same BPLs for three apps, and our middleware with different BPLs for three apps. We then check the battery level and memory usage (by reading `/proc/meminfo` file).

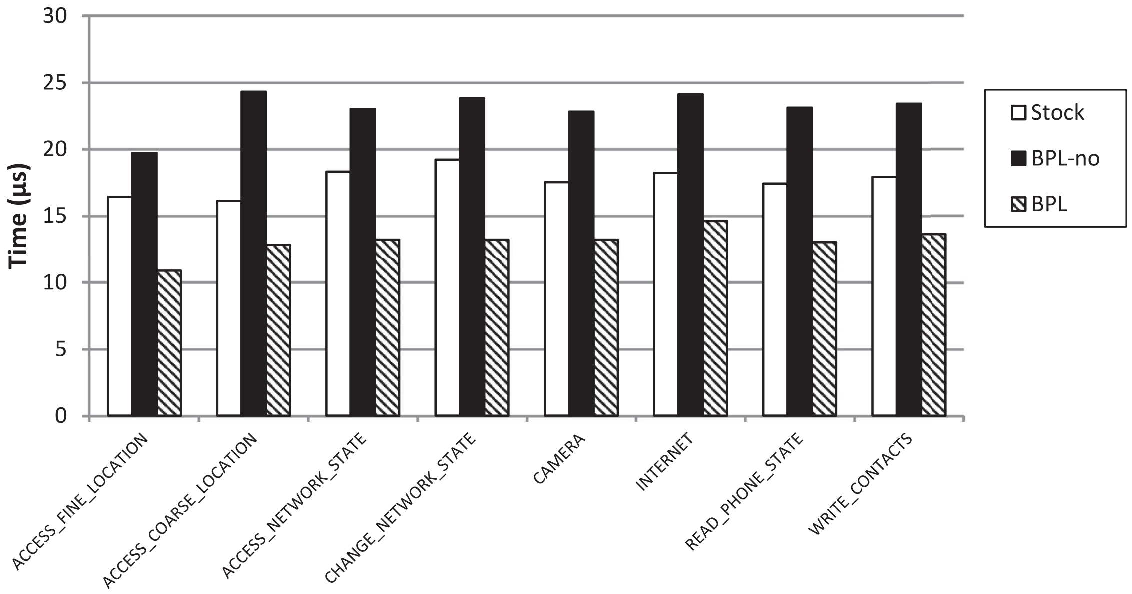

Fig. 3. Checking time of eight permissions under three settings.

Experimental results show that the battery usages of the stock Android and our middleware with the same BPLs are equal, while our middleware with the same BPLs increases the memory usage from 50.8 to 51.6 percent. In addition, our middleware with different BPLs uses 1 percent more battery and introduces 2.6 percent additional memory usage than the stock Android. The additional battery consumption and memory usage is mainly due to managing the multiple instances of frequently used camera apps.

8 Related Work

In this section, we categorize the related works and compare our work with them.

8.1 Solutions Against Permission Leaks

Permission leaks are common problems on Android which have been addressed in the previous works from different perspectives. Quire [37] annotates each inter-process communication (IPC) with the entire call chain to defend against the permission leaks. However, Quire is not transparent to app developers because it requires apps to be revised for annotating the IPC. Without requiring app modifications, XManDroid [38] addresses the issues by performing run-time monitoring and control on the communication links in Android's middleware and Linux kernel according to pre-defined security policies. Compared to XManDroid, our middleware is specifically designed for Android permission managers and it incurs a minimal impact on the usability of apps. Our work is similar to the IPC Inspection [11]. The main idea of IPC Inspection is to create a new app instance and reduce the permission list of receiver app when an ICC is invoked. Different from the IPC Inspection, our middleware creates and manages app instances according to the blocked permission lists of the involving apps.

8.2 Privilege Restriction

While Android permission managers provide a means to restrict the privileges of Android apps at runtime, there exists other privilege restriction techniques on Android. One example is Apex [42], which allows a user to selectively grant permissions at install time as well as define run-time constraints. Saint [43] is another solution which governs the install-time permission assignment and run-time usage according to certain app policies. Along this line, CRePE [44] enforces context-related policies on Android. CBAC [45] is a context based access control mechanism for Android users to set policies regulating the access of app to system resources and services under different contexts. DR BACA [46] leverages the context information and NFC technology to provide a dynamic role based access control for Android in a multi-user environment. MOSES [47] is a policy-based Android framework which enforces multiple isolated security profiles on a single device and dynamically switches from one profile to another under certain context. While these solutions improve the security of current Android's permission model, they rely on specific security policies and they are not specifically designed for Android permission managers.

Since Android allows apps to include third-party codes, such as advertising services and native code, another line of works focus on privilege restriction on such codes. Process-based privilege restriction is often used to separate third-party codes into different processes with different permissions. For example, AdSplit [48] and AFrame [49] separate the advertisements from apps and host them in processes to restrict their privileges. NativeGuard [50] is another solution leveraging the process-based protection in Android to isolate the native libraries in a permission limited process. Compared to them, our middleware addresses the generic problem of permission leaks which is not restricted to any third-party code.

8.3 Privacy Protection

Another line of research on Android security focuses on protecting user's privacy. AppFence [51], MockDroid [52] and TISSA [53] are three examples which enforce fine-grained access control on Android to protect user's privacy.

Instead of blocking the access to privacy data, they provide shadow data or mock data to untrusted apps and explore the trade-off between functionality and privacy. TaintDroid [54] is another example, which uses a dynamic taint analysis to prevent runtime attacks and data leakage. These works are complementary to our work and they can work side by side to enhance user privacy.

Recently, Shebaro et al. proposed IdentiDroid [55], which is a comprehensive solution for mobile users to guarantee its anonymity under anonymous network. In IdentiDroid, a data shadowing technique is used to hide the identifying information from apps, and a sensitive permission manager controls the sensitive permissions of apps at runtime. Our middleware can help address the permission leakage problem, which is not the focus of IdentiDroid.

8.4 Static Analysis on Application Codes

Permission leaks are mainly due to the existence of vulnerable app components. Much effort has been made to discover and address the vulnerability of Android apps or app components. CHEX [18] proposes to statically vet Android apps for detecting component hijacking vulnerabilities by tracking taints between externally accessible interfaces and sensitive sources or sinks. AppSealer [56] presents a technique to automatically generate patches for Android applications, which are subject to component hijacking vulnerability. RetroSkeleton [57] and SIF [58] are two rewriting frameworks that allow users to enforce high-level policies on Android apps. However, due to dynamic code loading techniques [59], it is not always possible for these solutions to detect all vulnerable components of apps.

9 Conclusion

It has been a major challenge to provide flexible and fine-grained permission control on Android apps. While various Android permission managers have been developed for this purpose, we discover that none of existing permission managers which we investigated enforce user-defined access control faithfully due to permission leaks. To address this problem, we propose a middleware which can be used by any existing permission manager. The middleware is designed to be (i) secure against any permission leaks; (ii) highly usable with minimum impact on the usability of any running app; (iii) transparent to app users and developers, and (iv) low in overall performance overhead (about 2.44 percent) and power consumption (about 1 percent). In the future, we plan to investigate how to help users configure appropriate BPLs for different types of apps in different contexts so as to enhance mobile security and user privacy in a user-friendly manner.

Acknowledgments

The authors thank the anonymous reviewers for their insightful comments. This paper was supported by National 973 Plan of China under grant No. 2014CB340600. This work was also supported by a research grant MX13C0094F from Huawei Technologies Co. Ltd. through Singapore Management University. Hai Jin is the corresponding author.

References

[1] IDC. (2013). Apple cedes market share in smartphone operating system market as android surges and windows phone gains [Online]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS24257413
[2] I. Pardines and V. Lopez, "Shop&go: TSP heuristics for an optimal shopping with smartphones," *Science China Inf. Sci.*, vol. 56, no. 11, pp. 1–12, 2013.
[3] Apple. (2014). Understanding privacy and location services on iphone, ipad, and iPod touch with iOS 8 [Online]. Available: http://support.apple.com/en-us/HT6338
[4] LBE. (2014). LBE security guard [Online]. Available: https://play.google.com/store/apps/details?id=com.lbe.security&hl=en
[5] SteelWorks. (2014). Advanced permission manager [Online]. Available: https://play.google.com/store/apps/details?id=com.gmail.heagoo.pmaster&hl=en
[6] M. Bokhorst. (2014). XPrivacy—The ultimate, yet easy to use, privacy manager [Online]. Available: https://github.com/M66B/XPrivacy
[7] CyanogenMod. (2014). cyanogenmod [Online]. Available: http://www.cyanogenmod.org/
[8] Xiaomi. (2014). MIUI [Online]. Available: http://en.miui.com/
[9] Huawei. (2014). Huawei Ascend P6 [Online]. Available: http://consumer.huawei.com/minisite/ascendp6/#home_landing
[10] R. Amadeo. (2013). App Ops: Android 4.3's hidden app permission manager, control permissions for individual apps! [Online]. Available: http://www.androidpolice.com/2013/07/25/app-ops-android-4-3s-hidden-app-permission-manager-control-permissions-for-individual-apps/
[11] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. 20th USENIX Conf. Security*, 2011, p. 22.
[12] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proc. 19th Annu. Symp. Netw. Distrib. Syst. Security*, 2012.
[13] P. P. Chan, L. C. Hui, and S. M. Yiu, "Droidchecker: Analyzing android applications for capability leak," in *Proc. 5th ACM Conf. Security Privacy Wireless Mobile Netw.*, 2012, pp. 125–136.
[14] X. Jiang. (2012). Smishing vulnerability in multiple android platforms [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/smishing.html, 2012.
[15] X. Jiang. (2012). SEND_SMS capability leak in android open source project (AOSP) [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/send_sms_leak.html
[16] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2013, pp. 623–634.
[17] Curesec. (2014). CVE-2013-6272 [Online]. Available: http://blog.curesec.com/article/blog/35.html
[18] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Security*, 2012, pp. 229–240.
[19] B. X-team. (2014). CVE-2014-8610 SMS resend vulnerability [Online]. Available: http://xteam.baidu.com/?p=164
[20] Google. (2014). Android open source project [Online]. Available: https://source.android.com/
[21] Google. (2014). Application fundamentals [Online]. Available: http://developer.android.com/guide/components/fundamentals.html
[22] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-application communication in android," in *Proc. 9th Int. Conf. Mobile Syst., Appl. Services*, 2011, pp. 239–252.
[23] Google. (2014). Android security overview [Online]. Available: https://source.android.com/devices/tech/security/
[24] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "Intentfuzzer: Detecting capability leaks of android applications," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Security*, 2014, pp. 531–536.
[25] L. Li, A. Bartel, J. Klein, and Y. le Traon, "Automatically exploiting potential component leaks in android applications," in *Proc. IEEE 13th Int. Conf. Trust, Security Privacy Comput. Commun.*, Sep. 2014, pp. 388–397.
[26] S. AppGuard. (2014). SRT AppGuard [Online]. Available: http://www.srt-appguard.com/en/
[27] permission manager. (2014). Permission Manager [Online]. Available: https://play.google.com/store/apps/details?id=com.gmail.permissionmanager&hl=en

[28] Xposed. (2014). App Settings [Online]. Available: http://repo.xposed.info/module/de.robv.android.xposed.mods.appsettings
[29] CollegeDev. (2014). DonkeyGuard [Online]. Available: https://play.google.com/store/apps/details?id=eu.donkeyguard
[30] Tencent. (2014). Tencent Mobile Manager [Online]. Available: https://play.google.com/store/apps/details?id=com.tencent.qqpimsecure
[31] Offshore. (2014). App Ops Starter [Online]. Available: https://play.google.com/store/apps/details?id=com.schurich.android.tools.appopsstarter&hl=en
[32] Appaholics. (2014). Permission Manager [Online]. Available: https://play.google.com/store/apps/details?id=com.appaholics.applauncher&hl=en
[33] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 627–638.
[34] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to android," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2012, pp. 274–277.
[35] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proc. 28th Annu. Comput. Security Appl. Conf.*, 2012, pp. 31–40.
[36] Chainfire. (2015). SuperSU [Online]. Available: https://play.google.com/store/apps/details?id=eu.chainfire.supersu&hl=en
[37] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proc. 20th USENIX Conf. Security*, 2011.
[38] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *Proc. 19th Annu. Netw. Distrib. Syst. Security Symp.*, 2012.
[39] AnTuTu. (2014). AnTuTu Benchmark [Online]. Available: http://www.antutu.com/en/index.shtml
[40] AnTuTu. (2014). AnTuTu Benchmark [Online]. Available: https://play.google.com/store/apps/details?id=com.antutu.ABenchMark
[41] Google. (2014). monkeyrunner [Online]. Available: http://developer.android.com/tools/help/monkeyrunner_concepts.html
[42] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending android permission model and enforcement with User-defined runtime constraints," in *Proc. 5th ACM Symp. Inf., Comput. Commun. Security*, 2010, pp. 328–332.
[43] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich Application-centric security in android," *Security Commun. Netw.*, vol. 5, no. 6, pp. 658–673, 2011.
[44] M. Conti, V. T. N. Nguyen, and B. Crispo, "Crepe: Context-related policy enforcement for android," in *Proc. 13th Int. Conf. Inf. Security*, 2011, pp. 331–345.
[45] B. Shebaro, O. Oluwatimi, and E. Bertino, "Context-based access control systems for mobile devices," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 2, pp. 150–163, Mar. 2015.
[46] F. Rohrer, Y. Zhang, L. Chitkushev, and T. Zlateva, "DR BACA: Dynamic role based access control for android," in *Proc. 29th Annu. Comput. Security Appl. Conf.*, 2013, pp. 299–308.
[47] G. Russello, M. Conti, B. Crispo, and E. Fernandes, "Moses: Supporting operation modes on smartphones," in *Proc. 17th ACM Symp. Access Control Models Technol.*, 2012, pp. 3–12.
[48] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating smartphone advertising from applications," in *Proc. USENIX Security Symp.*, 2012, pp. 553–567.
[49] X. Zhang, A. Ahlawat, and W. Du, "AFrame: Isolating advertisements from mobile applications in android," in *Proc. 29th Annu. Comput. Security Appl. Conf.*, 2013, pp. 9–18.
[50] M. Sun and G. Tan, "Nativeguard: Protecting android applications from Third-party native libraries," in *Proc. ACM Conf. Security Privacy Wireless Mobile Netw.*, 2014, pp. 165–176.
[51] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 639–652.
[52] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proc. 12th Workshop Mobile Comput. Syst. Appl.*, 2011, pp. 49–54.
[53] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Proc. 4th Int. Conf. Trust Trustworthy Comput.*, 2011, pp. 93–107.
[54] 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. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 1–6.
[55] B. Shebaro, O. Oluwatimi, D. Midi, and E. Bertino, "IdentiDroid: Android can finally wear its anonymous suit," *Trans. Data Privacy*, vol. 7, no. 1, pp. 27–50, Apr. 2014.
[56] M. Zhang and H. Yin, "Appsealer: Automatic generation of Vulnerability-specific patches for preventing component hijacking attacks in android applications," in *Proc. 21th Annu. Netw. Distrib. Syst. Security Symp.*, 2014.
[57] B. Davis and H. Chen, "RetroSkeleton: Retrofitting android apps," in *Proc. 11th Int. Conf. Mobile Syst., Appl. Services*, 2013, pp. 181–192.
[58] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "SIF: A selective instrumentation framework for mobile applications," in *Proc. 11th Int. Conf. Mobile Syst., Appl. Services*, 2013, pp. 167–180.
[59] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *Proc. 20th Annu. Netw. Distrib. Syst. Security Symp.*, 2014.

Daibin Wang is currently working toward the PhD degree in the school of computer science and technology at Huazhong University of Science and Technology, Wuhan, China. His research interests include mobile security and cloud security.

Haixia Yao has been a research engineer at Singapore Management University (SMU) since 2014. Before joining SMU, she was a research officer at the Institute of Infocomm Research (I2R) in Singapore for five and a half years. Over the years, she has been involved in many research and development projects, including mobile system development, hospital IT system development, enterprise IT system development, intelligent energy system development, secure e-mail system development, IDD call system development, and network administration software development.

Yingjiu Li is currently an associate professor in the School of Information Systems, Singapore Management University. His research interests include RFID Security and privacy, mobile and system security, applied cryptography and cloud security, and data application security and privacy. He has published more than 130 technical papers in international conferences and journals, and served in the program committees for over 80 international conferences and workshops. He is a senior member of the ACM and a member of the IEEE Computer Society. The URL for his web page is http://www.mysmu.edu/faculty/yjli/.

Hai Jin received the PhD degree in computer engineering from Huazhong University of Science and Technology (HUST) in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering at HUST in China. He is currently the dean in the School of Computer Science and Technology, HUST. In 1996, he received a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He was at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He received Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist in ChinaGrid, the largest grid computing project in China, and the chief scientists in National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He has coauthored 15 books and published more than 500 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a senior member of the IEEE and a member of the ACM.

Deqing Zou is a professor in the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST). He is responsible for system security research group of "Services Computing Technology and System Key Lab, MoE" and "Cluster and Grid Computing Key Lab, Hubei Province" at HUST, China. He has been the project manager of "863" project, NSFC project, security special project of National Development and Reform Commission, and so on. He was also the key member of "973" project, NSFC important research plan, and so on. He has applied for almost 20 patents, and published two books and more than 50 SCI/EI indexed papers, including paper published in *IEEE Transactions on Dependable and Secure Computing*. He got one piece of the first-class Science & Technology Progress Award (Hubei Province) and one piece of the first-class technical invention award (MoE). He is on the editorial boards of five international journals, and has served as PC chairs /members of more than 40 international conferences.

Robert H. Deng has been a professor at the School of Information Systems, Singapore Management University since 2004. Prior to this, he was the principal scientist and manager in the Infocomm Security Department, Institute for Infocomm Research, Singapore. His research interests include data security and privacy, multimedia security, network and system security. He was an associate editor of the *IEEE Transactions on Information Forensics and Security* from 2009 to 2012. He is currently an associate editor of the *IEEE Transactions on Dependable and Secure Computing*, and member of Editorial Board of the *Journal of Computer Science and Technology* (the Chinese Academy of Sciences), and the *International Journal of Information Security (Springer)*, respectively. He is the chair in the Steering Committee of the ACM Symposium on Information, Computer and Communications Security (ASIACCS). He received the University Outstanding Researcher Award from the National University of Singapore in 1999 and the Lee Kuan Yew fellow for Research Excellence from the Singapore Management University in 2006. He was named Community Service Star and Showcased Senior Information Security Professional by (ISC)2 under its Asia-Pacific Information Security Leadership Achievements program in 2010.