Journal of Computer Security 24 (2016) 405–426
DOI 10.3233/JCS-160552
IOS Press

TRIPLEMON: A multi-layer security framework for mediating inter-process communication on Android

Yiming Jing [a], Gail-Joon Ahn [a,*], Hongxin Hu [b], Haehyun Cho [a] and Ziming Zhao [a]
[a] *Ira A. Fulton School of Engineering, Arizona State University, Tempe, AZ 85287, USA*
E-mails: ymjing@asu.edu, gahn@asu.edu, haehyun.cho@asu.edu, zmzhao@asu.edu
[b] *School of Computing, Clemson University, Clemson, SC 29634, USA*
E-mail: hongxih@clemson.edu

Abstract. As smartphones have become an indispensable part of daily life, mobile users are increasingly relying on them to process personal information with feature-rich applications. This situation requires robust security mechanisms for protecting sensitive applications and data on mobile devices. Android, as one the most popular smartphone operating systems, provides two core security mechanisms, *application sandboxing* and a *permission system*. However, recent studies show that these mechanisms are vulnerable to be passed by a variety of attacks. In this paper, we argue for the need of designing and implementing more comprehensive security mechanisms for Android. We realize that mediating Inter-Process Communication (IPC) channels used by Android applications can mitigate prominent attacks effectively and efficiently. Based on this observation, we propose a practical multi-layer security framework called TRIPLEMON to support policy-based mediation on Android IPC. We also discuss and evaluate a proof-of-concept prototype of TRIPLEMON along with the experimental results derived from real malware samples and synthetic attacks.

Keywords: Mobile security, system security, Android, reference monitor, mandatory access control

1. Introduction

According to IDC [31], the worldwide smartphone market is expected to grow 32.7% year over year in 2013, reaching a total shipment of 958.8 million units. With the surging computing power and network connectivity of smartphones, more applications are deployed on smartphone platforms to enable users to customize their devices. Meanwhile, users are increasingly relying on applications to process their personal information. However, many applications tend to manipulate users' sensitive information such as contact list, location information, and other credentials. This requires robust security mechanisms for mobile platforms to protect sensitive resources.

Android, as a fast-evolving modern smartphone operating system, implements a security architecture that adopts a kernel-level application sandbox to ensure that applications can only access their own files, and a permission system to control accesses on system resources. Although these security mechanisms significantly confine applications' privileges, we have seen a number of attacks that can bypass them. We classify these attacks as two categories: (1) *attacks bypassing permissions*. A malicious application may

*Corresponding author. E-mail: gahn@asu.edu.

abuse the components of other applications to gain extra privileges via Inter-Component Communication (ICC) [5,15,21,26]. Moreover, irrevocable and coarse-grained permissions grant unnecessary capabilities to applications, which violates the principle of least privilege [25]; (2) *attacks bypassing the sandbox*. Sensitive resources of system services can be accessed by any application via non-permission-protected APIs or globally readable files [28]. In addition, malicious applications can exploit vulnerabilities in the device nodes and Linux kernel to gain root privileges [24,50].

To defend Android against these attacks, a number of security extensions have been proposed to implement enhanced mandatory access control (MAC) in Android. Some frameworks implement enforcement mechanisms by modifying sensitive Android APIs [1,3,4,39,44,51]. Given the large number of Android APIs, these frameworks would require extensive work to achieve complete mediation. Meanwhile, some frameworks implement reference monitors by rewriting applications [16,33,40,43,47]. However, a recent study [29] shows that such frameworks may not be tamper proof.

In this paper, we re-explore the problem of designing and implementing a practical MAC framework for Android. We propose a multi-layer security framework called TRIPLEMON. Unlike previous work, TRIPLEMON opts for a system-centric and IPC-oriented approach. Specifically, TRIPLEMON extends the IPC subsystem of Android to mediate various types of IPC channels that can be used by Android applications to access resources outside their sandboxes. While supporting complete mediation and being tamperproof, TRIPLEMON also enables users to flexibly control the capabilities of applications in a fine-grained manner. We have implemented TRIPLEMON as a combination of three reference monitors and a decision manager. To evaluate the effectiveness of TRIPLEMON, we perform a large-scale analysis on malware samples and popular benign applications to reveal their behaviors regarding IPC. Afterwards, we show that TRIPLEMON can effectively protect sensitive resources against a variety of prominent attacks.

This paper makes the following contributions:

- We discuss the challenges in enabling effective and practical mandatory access control in Android;
- We propose a multi-layer security framework, TRIPLEMON, to address recent prominent attacks at different layers of the Android software stack. TRIPLEMON enables policy-driven mandatory access control tailored to the peculiarities of Android IPC;
- We implement a proof-of-concept prototype of TRIPLEMON and demonstrate how it can be seamlessly deployed in Android; and
- We evaluate TRIPLEMON with comprehensive experiments. Our experimental results demonstrate the feasibility and practicability of TRIPLEMON.

The remainder of this paper proceeds as follows. Section 2 describes two classes of existing security issues in Android and existing countermeasures. Section 3 explains the Android IPC mechanisms. Section 4 introduces the design of TRIPLEMON. Section 5 elaborates the evaluation results. Section 6 discusses the limitation of our approach and future work. Section 7 describes related work. Section 8 concludes this paper.

2. Motivation

In this section, we first discuss the attacks that bypass existing Android security mechanisms. We then articulate the existing countermeasures that defend Android against such attacks.

2.1. Existing attacks

Android has been a promising mobile platform nowadays. It includes a monolithic Linux kernel and a loosely-coupled middleware layer. Due to the mixed nature of this platform, Android not only inherits security issues rooted in the Linux kernel, but also faces unique security problems introduced by the middleware.

2.1.1. Attacks bypassing permissions

A malicious application may perform privilege escalation attacks by abusing ICC. As applications may unintentionally expose private components without proper permission protection, ICC can be abused for unauthorized intent receipt and intent spoofing [5]. For example, a malicious application can sniff, modify and replace messages sent by a benign application. Moreover, public components are also vulnerable to capability leaks. Lineberry et al. shows that a zero-permission application may upload without the `INTERNET` permission by sending an intent to the Browser application [17]. Recently researchers have identified several similar zero-permission attacks, such as making phone calls [22] and playing an alarm [26].

A coarse-grained permission covers multiple capabilities for accessing resources with different sensitivity levels. For example, `READ_PHONE_STATE` allows an application to read both phone's state (e.g., if there is an incoming call) and phone's unique identities [20]. The former is trivial but the latter indeed introduces potential privacy risks. For applications that utilize the phone's state only, holding unnecessary capability like reading the phone's unique identities violates the principle of least privilege. In addition, capabilities granted to applications via Android permission system are irrevocable. In other words, once an application is granted a permission, the user has no way to revoke partial capabilities of the granted permission.

System services may fail to protect their APIs with appropriate permissions, exposing sensitive resources to all applications. For example, the `RingerMode` setting of AudioService is not protected by any permission, allowing a malicious application to silent the phone without user consent. As addressed in [23], only 6.45% of all API methods in Android are protected by at least one permission.

2.1.2. Attacks bypassing the sandbox

Files that are set globally accessible render DAC-based isolation useless. For example, a vulnerability in Skype exposes the user's profile information and instant messages due to improper file permission settings [10]. Similarly, the file that stores the list of applications is globally readable in Android and any application can access it without extra permission [28].

A malicious application may also exploit Linux kernel vulnerabilities to break the sandbox [7–9,11–13]. These exploits (also known as "jailbreaks") allow adversaries to escalate their privilege to root privilege and bypass existing security mechanisms. Recent reports have shown the existence of root-capable malicious applications that download additional malware [41] and replace system binaries [42]. Indeed, the occurrence of such exploits is quite high [50] and almost every popular Android device had a publicly available exploit for at least 74% of the device's lifetime [24].

2.2. Existing countermeasures

Given the diversified attacks, it is imperative to remedy Android's default security mechanisms to provide better security guarantees. Recently, a wide spectrum of security extensions has been proposed to implement enhanced MAC in Android. Depending on how they implement MAC, we can divide them into two categories.

2.2.1. MAC implemented in the Android system

The history of SELinux has demonstrated the effectiveness of system-centric MAC implementations. Such MAC implementations require patching and/or recompiling the system. In recent years, plenty of security frameworks [1,3,4,6,38,44,51] follow this approach to (1) add kernel MAC for reinforcing application sandboxes; and/or (2) add middleware MAC to remedy the shortcomings of permissions. The kernel MAC implementations reuse or extend previous Linux MAC frameworks such as SELinux and TOMOYO Linux. The middleware MAC implementations are usually tailored to the problems they attempt to address. In particular, FlaskDroid [4] is the first security extension that attempts to address diverse security requirements with a generic security framework. However, generic MAC implementations come with a cost of relatively large code base and difficulties in maintainability. For example, FlaskDroid includes 12 Userspace Object Managers (USOM). Each USOM must be re-evaluated and patched once new Android versions are released. And it remains a question whether the 12 USOMs can completely cover all the attack vectors. Android Security Framework (ASF) [1] attempts to address the limitations of FlaskDroid with loadable security modules and a comprehensive set of security APIs. Despite that the loadable modules could reduce the work required to instantiate different security models, the security APIs themselves that are scattered in various Android system services still suffer from deployment and maintenance issues.

2.2.2. MAC implemented in applications

To address the deployment issues of system-centric MAC implementations, application-centric approaches [16,33,40,43,47] have been proposed. These security frameworks, which are also known as inlined reference monitors, rewrite applications by instrumenting applications' byte code or native code and thus do not require any changes to the underlying Android system. Despite their significant improvement in deployment, a recent study [29] shows that application-centric MAC implementations are subtle and they could be bypassed or subverted, because the instrumented code runs within the same process as that of the confined application. Apparently, an application is able to modify itself to remove or suppress the instrumented code.

3. Android IPC

While the Android application sandbox disallows an application to directly access the resources of other applications and Android system, the application can communicate with other applications and utilize the system services through several Android IPC channels. Unfortunately, these IPC channels can be abused by malicious applications (see examples in Section 2). In this section, we elaborate three kinds of Android IPC mechanisms in detail, which will serve as the basis for our approach. Figure 1 depicts several examples of Android IPC channels.

3.1. Inter-component communication

Inter-Component Communication (ICC) is commonly used to support communication between Android applications. Android applications are compartmentalized into individual components, including activities, services, broadcast receivers, and content providers. An application's components can be published and accessed by other applications via ICC channels. Specifically, an application can request an ICC channel with a message called *intent*. An intent holds the description of the caller's expected

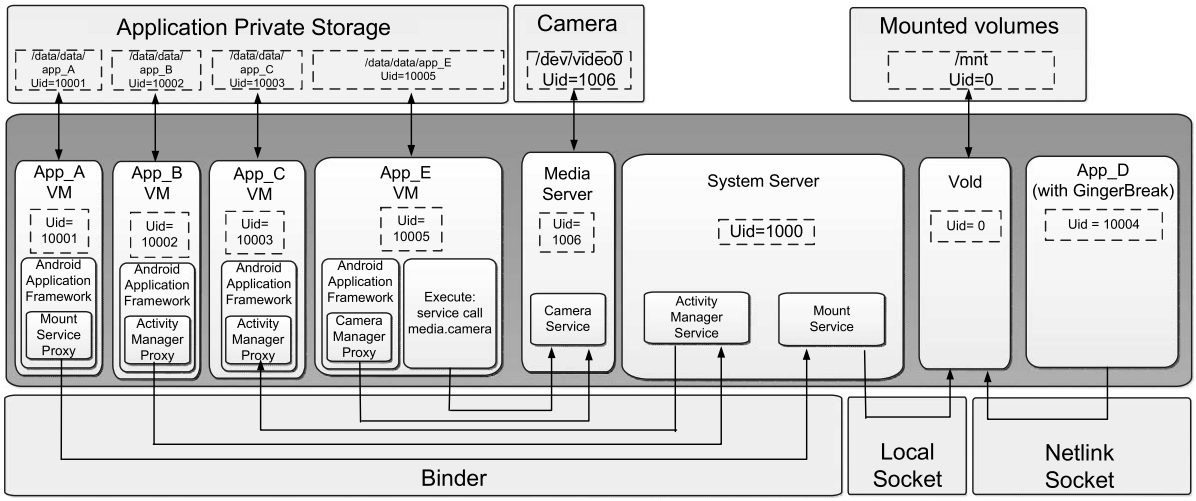

Fig. 1. Examples of Android inter-process communication.

functionalities, *e.g.*, viewing a picture. Then, the intent is handled by a system service called ActivityManagerService, which delivers the intent to valid recipient application components. As shown in Fig. 1, application `B` and application `C` utilize ICC to communicate with each other. The arrows show that application `B` sends an intent to application `C` via ActivityManagerService.

3.2. Binder IPC

Applications frequently access system resources using Android APIs, which are implemented with Binder IPC channels established between applications and system services. A Binder IPC channel is established through the following six steps: (1) an application invokes an API that is essentially a public function of a proxy in the application framework; (2) the proxy encapsulates the service's handle, the caller's data, and a command code in a parcel; (3) the proxy sends the parcel to Binder; (4) Binder delivers the parcel to the remote service; (5) the remote service unpacks the parcel, handles the data as instructed by the command code, and sends results back to the proxy via Binder; and (6) the application receives the results. Alternatively, an application can send Binder messages without involving proxies of the application framework. In such a case, the application can use a utility called `service`, which is a universal proxy, to initiate Binder IPC with any service. In addition, Binder maintains and protects the identities (UID and PID) of caller and callee processes. A service can also get the UID of the original caller by querying Binder.

As depicted in Fig. 1, application `E` can utilize the camera via the CameraManagerProxy of the application framework or via a tool called `service`. Either approach leverages Binder IPC to communicate with CameraService.

3.3. Linux IPC

Traditional Linux IPC mechanisms are still available in Android. Examples include local sockets, Unix-domain sockets, and Netlink sockets. In a broader sense, Linux IPC also includes communications via signals and files. System services can use Linux IPC to communicate with other services and the kernel. At the same time, applications may communicate with system services via Linux IPC, bypassing

Android's middleware. Note that applications by default cannot directly utilize Linux IPC channels created by privileged system processes, but recent attacks [48,49] have demonstrated the feasibility of exploiting globally accessible resources.

Figure 1 shows that application `A` accesses mounted volumes (e.g., unmounting the SD Card). MountService communicates with Vold through a local socket. The local socket has the following parameters: file permission (`660`), UID (`root`), and GID (`mount`). Thus, applications are restricted from accessing Vold directly. However, System Server can access Vold because it is assigned to `mount` group. In addition, Vold exposes a Netlink socket to receive events (e.g., unplugging storage media) from the kernel. Unfortunately, these events can be forged by an unprivileged and malicious application, as demonstrated in the GingerBreak exploit [11,12].

4. System design

In this section, we first discuss the design goals in implementing an effective and practical MAC framework for Android. We then propose our design and implementation of TRIPLEMON that is tailored to the peculiarities of Android IPC.

4.1. Design goals

Mandatory access control is a type of access control by which the operating system confines the abilities of subjects to access certain objects based on a centralized security policy. An effective MAC implementation, no matter whether it is system-centric or application-centric, should fulfill at least the following design goals:

G1 *Complete mediation*. We should implement mechanisms that completely mediate access vectors that could be used by a subject in the system, so that the subject cannot bypass our mechanisms.

G2 *Tamper proof*. We should prevent attackers from undermining our security mechanisms. This includes protecting the integrity of the security mechanisms themselves as well as protecting security policies.

Towards a practical MAC implementation, we further propose the additional design goals:

G3 *Generic and flexible*. We should be able to dynamically re-configure the security mechanisms using flexible security policies.

G4 *Unified policy scheme*. Composing and managing security policies could be tedious and error-prone for system administrators. To reduce the workload of policy management, we expect the security policy to use a unified scheme for different types of subjects, objects, and operations.

G5 *Easy maintenance*. The maintenance of the code should be simple across different OS versions. In particular, it should require little work to re-evaluate the effectiveness of the MAC implementation against newer OS versions.

We assume a threat model that the Android system is trusted. In other words, our trusted computing base (TCB) includes the Linux kernel and the Android middleware, and we do not trust any installed Android application.

4.2. Overview

TRIPLEMON is a system-centric, multi-layer, and policy-agnostic MAC implementation that fulfills our proposed design goals. Figure 2 depicts the architecture of TRIPLEMON, which consists of a set of

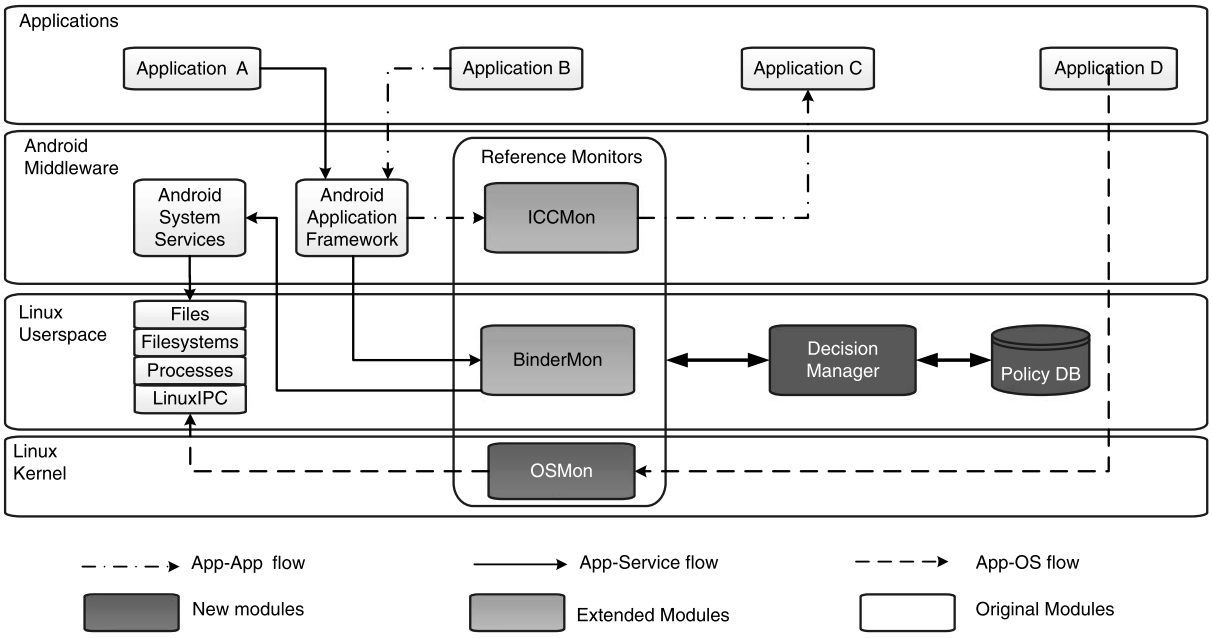

Fig. 2. TRIPLEMON architecture.

reference monitors and a decision manager. TRIPLEMON opts to a multi-layer design because Android itself is composed of three layers: applications, middleware, and Linux. Each layer has its respective access control semantics and thus requires a dedicated reference monitor.

TRIPLEMON's reference monitors use a different design compared with FlaskDroid. FlaskDroid achieves G3 by placing enforcement hooks in various Android system services, such as ActivityManager, SensorManager, and TelephonyManager. However, it is hard to justify the completeness (G1) of such an approach, because it is infeasible and futile to hook every function in more than 50 system services in Android. And maintaining various hooks here and there is also difficult in a fast evolving operating system like Android (G5). To address this issue, we are inspired by SELinux which uses few hooks that are actually "choke points" of sensitive operations. In particular, we identify the boundaries of IPC channels as our choke points because Android applications *must* use IPC to access sensitive resources outside their sandboxes. Furthermore, Android IPC mechanisms are part of the core libraries and they have not been changed remarkably compared to the system services that FlaskDroid depends on. Therefore, mediating ICC, Binder IPC, and Linux IPC channels allows TRIPLEMON to minimize the cost of G5 while satisfying G1 and G3. Moreover, the reference monitors at lower layers (OSMon) can protect the integrity of reference monitors at higher layers (BinderMon and ICCMon).

TRIPLEMON also includes a decision manager to address the semantic gaps among reference monitors. Semantic gaps arise when a resource corresponds to multiple objects at different layers. For example, an application can take a picture using (1) the camera application; (2) the system service that controls the camera; or (3) the device node of the camera (`/dev/video0`). Suppose we are to revoke an application's capabilities of taking pictures, the decision manager can issue decisions for all the reference monitors to block accesses at all layers. In addition, the decision manager also follows a unified policy scheme and resolves inconsistencies once conflicting access control decisions are generated for

the same object. The unified policy scheme also makes it easier to analyze security policies at different layers to evaluate the system-wide assurance.

4.3. Multi-layer reference monitors

In this section, we elaborate the design of the three reference monitors and explain how they achieve our proposed design goals.

4.3.1. ICCMon

We propose ICCMon to handle ICC requests that are between applications. We choose ActivityManagerService as the choke point for ICCMon because ActivityManagerService is the single system service responsible for establishing and shutting down ICC channels.

Figure 3 depicts the workflow of ICCMon. Specifically, ICCMon labels application components with the UIDs of their corresponding application processes, and labels intents with their meta-information including action, category, and data URI. To ensure complete mediation, we analyze the call graph of ActivityManagerService and identify 4 functions that can capture all ICC requests, including `startActivityLocked()`, `startServiceLocked()`, `getContentProvider()` and `deliverToRegisteredBroadcastReceiversLocked()`. From the names of these functions we can notice that they handle ICC channels to activities, services, broadcast receivers, and content providers, respectively. These hooks placed just before ActivityManagerService is about to establish an ICC channel, which is similar to how hooks are placed in Linux Security Modules. Using these hooks, ICCMon enforces security decisions acquired from the decision manager. ICCMon returns control to ActivityManagerService if an ICC request is accepted. Otherwise, ICCMon generates a security exception for the caller application to shut down the ICC channel.

4.3.2. BinderMon

Android APIs are implemented based on Binder IPC. The mappings between Android APIs and Binder IPC transactions are specified in the AIDL[1] files provided with the source code of Android. For example, Binder IPC requests on *whose destination is a system service called* `iphonesubinfo` *and*

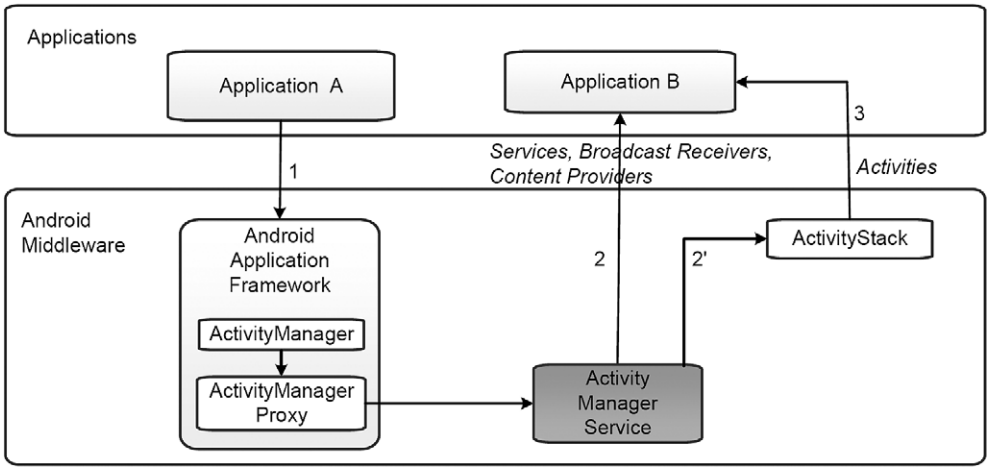

Fig. 3. ICCMon workflow.

[1] Android Interface Definition Language.

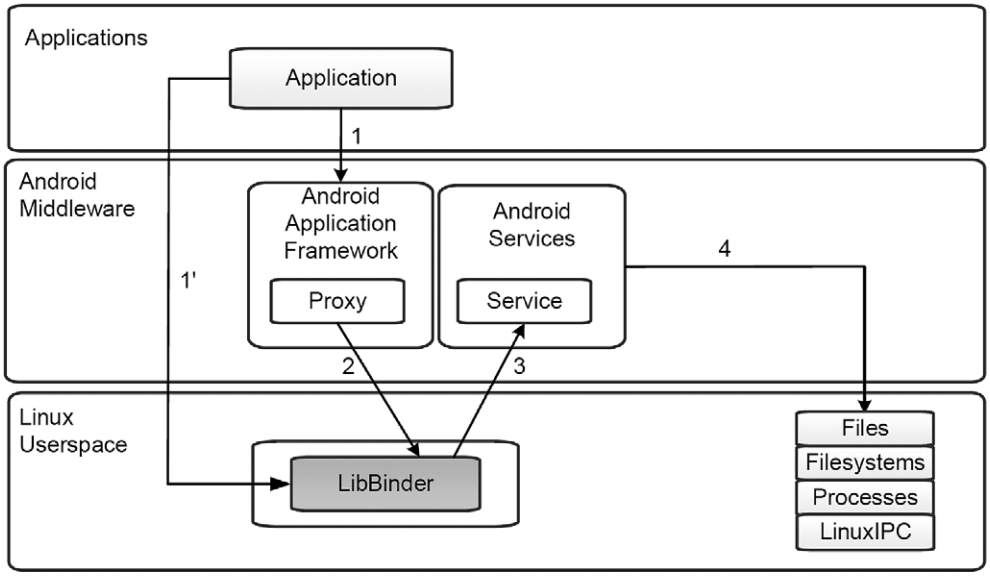

Fig. 4. BinderMon workflow.

whose command code is 1 correspond to an API `getDeviceID`. Therefore, we can infer the semantics of any Binder IPC request by checking corresponding Android API that it can be mapped to.

To identify choke points of Binder IPC, we analyze the entire Binder IPC subsystem and we choose to place the enforcement hooks in LibBinder (`/system/lib/libbinder.so`), because it is the dynamic library linked to all the system services. As long as LibBinder and the system services are not compromised, BinderMon can intercept all the Binder IPC requests sent to the system services.

BinderMon mediates Binder IPC before the permission framework is consulted. Therefore, it enables dynamic permission revocation without affecting the existing permission framework. Furthermore, BinderMon offers finer granularities at the API level, whereas the granularity of the Android permission framework is a set of permissions where each permission maps to multiple APIs. Moreover, BinderMon protects 1,448 public and private Android APIs implemented by more than 70 system services, while FlaskDroid only protects 136 APIs in 12 system services.

A detailed workflow of BinderMon is shown in Fig. 4. To make an API call, an application sends an Binder IPC request using the proxy of the API's corresponding remote system service (Step 1). Alternatively, this application can send a request without involving the proxy (Step 1′). BinderMon intercepts both types of requests and queries the decision manager with the caller application's UID, the callee service's name, and the command code which indicates the API to be called. If the request is allowed, BinderMon returns control to LibBinder. Otherwise, BinderMon shuts down the IPC channel and returns an error code `PERMISSION_DENIED` to the caller application.

4.3.3. OSMon

The default Android application sandbox is flawed. Although Android applications are expected to use Android APIs to access system resources, they still possess capabilities to bypass APIs and access sensitive Linux IPC channels that general Android applications should not necessarily use. However, ICCMon and BinderMon cannot revoke all of these unnecessary capabilities. Therefore, we propose a kernelspace reference monitor, OSMon, to mediate Linux IPC channels and enforce the principle of least privilege for Android applications.

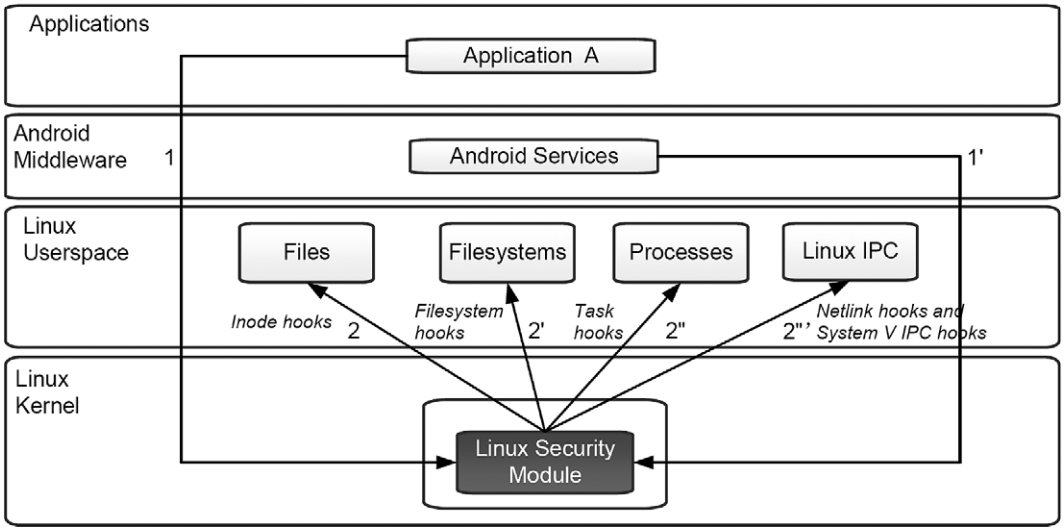

Fig. 5. OSMon workflow.

Table 1
Implemented hooks in OSMon

Hook category	Hook name
Local socket	socket_create, socket_connect, socket_bind, socket_send
Unix-domain socket	ud_connect, ud_send
Netlink socket	netlink_send, netlink_recv
Task	task_create, task_setuid, task_setgid, task_kill
File	inode_create, inode_rename, inode_mkdir, inode_rmdir, inode_link, inode_symlink, inode_unlink, inode_setattr, dentry_open
File system	sb_mount, sb_unmount

Figure 5 depicts the workflow of OSMon. Similar to SELinux [45] and TOMOYO Linux [30], we identify choke points as the hooks defined by Linux Security Module (LSM) [46]. However, OSMon only use the hooks are specific to the Linux IPC mechanisms. Table 1 shows the implemented 23 hooks which cover objects like inodes, file systems, tasks, local sockets, Unix-domain sockets, and Netlink sockets. These hooks can be enabled or disabled individually in the kernel configuration file. In addition, OSMon plays as the trusted computing base (TCB) of the entire TRIPLEMON framework. It protects all the userspace components of TRIPLEMON, including ICCMon, BinderMon, the decision manager, and the policy database. Moreover, OSMon protects itself against root exploits by depriving untrusted applications' capabilities to access IPC channels that may subvert OSMon.

4.4. Multi-layer policy modules

Our unified policy scheme for TRIPLEMON is defined as follows:

Definition 1 (Target)**.** A target is a 3-tuple (*Subject*, *Resource*, *Action*), where

- *Subject* is a set of entities to which the authorization is granted;
- *Resource* is a set of entities to which accesses need to be mediated; and
- *Action* is a set of actions being authorized or forbidden.

Table 2
Components in TRIPLEMON Policies

		ICC policy	Binder policy	OS policy
Target	Subject	Application group Application Component	Application group Application	Application group Application System File
	Resource	Application group Application Component	Service	Linux IPC channel Process File Filesystem
	Action	startActivity bindService sendBoradcast accessContentProvider	Call	Linux IPC Task Inode Filesystem
Condition		Action Category Data	Service cmd code	System call's name Parameters
Effect		Accept Deny	Accept Deny	Accept Deny

Definition 2 (Access control policy)**.** An access control policy is a 3-tuple (*Target*, *Condition*, *Effect*), where

- *Target* decides whether an access request is applicable to the policy. The target specification is defined in Definition 1;
- *Condition* specifies restrictions on the attributes in the target and refines the applicability of the policy; and
- *Effect* $\in$ {*accept*, *deny*} is the authorization effect of the policy.

To evaluate an access request over access control policies, if the request satisfies both the target and condition of a policy, the response is sent with the decision specified by the effect element in the policy. Otherwise, the response yields "deny".

We next describe three kinds of TRIPLEMON policy: ICC policy, Binder policy and OS policy in detail. Table 2 summarizes the major components contained in three kinds of TRIPLEMON policies.

4.4.1. ICC policy

ICC policies regulate the intent-based IPC between applications. Thus, *application group*, *application*, and *component* comprise the `subject` and `resource` of an ICC policy. `Actions` correspond to the APIs that initiate ICC to four types of components, namely activities, services, broadcast receivers and content providers. `Condition` is defined as the attributes used in intents. Table 2 illustrates the elements of ICC policy.

For example, an adversary who does not possess the INTERNET permission may still access the Internet by sending an intent with action `ACTION_VIEW` and the URL to the Browser application. We can mitigate such a privilege escalation attack by specifying and enforcing an ICC policy to prevent the adversary from sending such an intent to the Browser application.

4.4.2. Binder policy

Binder policies specify the behaviors of Binder IPC channels between applications and system services. Thus, *application group* and *application* comprise the `subject`, and *system service* comprises the `resource`. `Action` has only one instance, *call*. And `condition` specifies the *command code* of the command to be executed by the remote services, as shown in Table 2.

For instance, as we discussed previously, the permission `READ_PHONE_STATE` allows an application to access resources such as the unique device IDs and the phone's state. We can set a Binder policy to revoke an application's privilege of accessing the device IDs but keep an access privilege to the phone's state without reinstalling the application. Compared with the default permission framework which treats the capabilities for a permission as an indivisible block, our approach enables a more fine-grained and revocable access control.

4.4.3. OS policy

OS policies mediate Linux IPC channels. In an OS policy, as illustrated in Table 2, `subject` consists of *application group*, *application*, *system* and *file*. `Resource` consists of *file*, *filesystem*, *process* and *Linux IPC channel*. `Action` has four types: *inode*, *filesystem*, *task*, and *Linux IPC*. These actions correspond to different categories of operations on Linux IPC channels. `Condition` specifies the name of the exact operation and optional parameters.

As we mentioned earlier, GingerBreak is a root exploit that attacks a system service called Vold by sending forged and malformed Netlink messages. Such an exploit can be easily prevented by defining an OS policy that disables applications from sending Netlink messages. Indeed, Android applications are currently not strictly prohibited from using some operating system features. Malicious applications may take advantage of these features to launch attacks. Thus, we need to define corresponding OS policies to prevent applications from abusing these features.

4.5. Decision manager

The decision manager is the policy decision point of TRIPLEMON where IPC requests are evaluated against authorization policies. It centrally issues access control decisions for each reference monitor, manages security policies, and resolves conflicts among reference monitors.

4.5.1. Communication with reference monitors

As the only policy decision point in TRIPLEMON, the decision manager runs in a dedicated process and communicates with the reference monitors via a local socket interface. The communication follows the policy scheme we defined in Section 4.4. Indeed, this interface needs special attention because it can subvert TRIPLEMON or cause denial of service if exploited. To protect this interface, OSMon enforces a set of top-priority policy rules to only allow the decision manager to write into the interface.

4.5.2. Policy management

The decision manager parses a JSON-like plaintext policy file that follows the policy scheme defined in Section 4.4. The policy file is stored in an internal read-only filesystem. In addition, OSMon enforces rules to disallow access on the policy database from any process except the decision manager.

The decision manager maintains two tracks of security policies, namely *slow track* and *fast track*. The slow track is enforced by OSMon only. This track defines the common and least privileges of general installed applications. The fast track is enforced by ICCMon and BinderMon. This track expects more frequent policy updates because ICCMon and BinderMon are required to meet per-application security requirements. Specifically, ICCMon or BinderMon always consults the decision manager to acquire

decisions generated based on the latest fast track policy. On the contrary, OSMon caches a copy of the slow track policy in the kernel and makes decisions by itself. It only consults the decision manager to get the latest slow track policy when it initializes itself.

4.5.3. Decision reconciliation

Decision reconciliation is necessary to resolve conflicts when multiple reference monitors are involved to mediate accesses on the same resource. For example, an adversary attempts to access the camera via ICC and Binder IPC, expecting that there would be a capability leak somewhere. Suppose a strategy, `deny-overrides`, is applied. The decision manager ensures that ICCMon and BinderMon both deny the requests to access the camera if there is any applicable ICC or Binder policy that evaluates to deny. More flexible strategies [34], such as `first-applicable` and `strong-consensus`, can be adopted to resolve the decision inconsistencies.

5. Evaluation

In this section we evaluate TRIPLEMON in terms of its coverage, effectiveness and performance overheads. Our experiments were performed on a Galaxy Nexus running Android 4.2.2 and Linux kernel 3.0.31.

5.1. Experimental setup

Similar to other policy-driven MAC implementations, TRIPLEMON requires a good security policy to be effective. In TRIPLEMON, we opt to semi-automatically and iteratively derive policy rules from applications' runtime behaviors. We first configured TRIPLEMON into a permissive mode where it only logs IPC requests. We then executed a set of benign applications and a set of malicious applications. By comparing the observed IPC requests, we identified the IPC requests to be allowed or denied in our security policy.

For the set of benign applications, we handpicked 10 applications (Table 3) from Google Play's top charts. These applications are from renowned developers and under different application categories. We assumed that these applications are trusted by general users and we used them to outline the general expected runtime behaviors of Android applications. The malicious applications were 1,260 malware samples from the Android Malware Genome Project [50]. We fuzzed each application through 5 iterations with randomized touch inputs and system events.

Table 3
Applications assumed to be benign and trusted by general users

Application	Category
AmazonMobile	Shopping
BejeweledBlitz	Game
ChaseMobile	Finance
Dictionary.com	Books & reference
Dropbox	Productivity
Google+	Social
GooglePlayMovies&TV	Media & video
Hangouts (replacesTalk)	Communication
MoviesbyFlixster	Entertainment
Yelp	Travel & local

Table 4
System services protected by TRIPLEMON but unprotected by FlaskDroid

Service	Example APIs
AccountManager	getAccounts, getPassword, peekAuthToken, invalidateAuthToken
AlarmManager	setTimeZone
BackupManager	setBackupEnabled, setAutoRestore
Bluetooth	createBond, isDiscovering, getUuids, getScanMode
ConnectivityManager	getActiveNetworkInfo, getProxy, tether, startUsingNetworkFeature
EmailService	searchMessages, loadAttachment, sendMeetingResponse
NFCManager	setForegroundDispatch, setNdefPushCallback
SipService	open, close, createSession, setRegistrationListener
VibrationService	vibrate, vibratePattern
WifiManager	setFrequencyBand, getWifiApConfiguratin, getScanResults

5.2. Coverage

The generated policy demonstrated the coverage of system resources protected by TRIPLEMON. Compared to one of closely related work FlaskDroid, TRIPLEMON provides the same level of protection on ICC channels and much more protection on Binder IPC channels. Table 4 shows the system services that appeared in TRIPLEMON's security policy but cannot be protected by FlaskDroid. This is due to the fact that FlaskDroid only addresses 12 system services [4]. To achieve the same protection level as TRIPLEMON, FlaskDroid would need to modify thousands of APIs scattered in more than 60 system services. This limitation in FlaskDroid is an inevitable problem inherited from its design and it does not meet the mediation requirements of reference monitor [32]. However, TRIPLEMON is capable of protecting more services by employing a generic approach, which implements hooks at the choke points of IPC and ICC.

In terms of Linux IPC channels, TRIPLEMON provides relatively less coverage compared to FlaskDroid whose kernel MAC is built upon SE Android. Indeed, TRIPLEMON only implements 23 hooks, as shown in Table 1, which are specific to the Linux IPC mechanisms. Even though TRIPLEMON cannot cover as many objects as SE Android does, it covers objects that are essential to Linux IPC mechanisms, such as inodes, tasks and sockets. However, we note that OSMon follows a policy schema that is consistent with BinderMon and ICCMon, which policy schemas follow eXtensible Access Control Markup Language (XACML) standard [37]. Meanwhile, with respect to FlaskDroid, its middleware policy does not use SE Android's policy language but introduces their own language that bears a resemblance to SELinux' language. Hence, it is hard to say that FlaskDroid satisfies the consistency of policy languages between middleware MAC and SE Android. A consistent policy schema is necessary for efficient policy management.

5.3. Case studies

To further validate the effectiveness of TRIPLEMON, we tested TRIPLEMON against real malware samples and synthetic applications that implement the attacks that we discussed in Section 2.

5.3.1. ICCMon vs information stealing malware

To test ICCMon, we selected a malware strain called Gone60 (short for "gone in sixty seconds"). Gone6 accesses the Contact applications via ICC channels and uploads user's contacts to remote servers. In our experiments, we put the malware samples into an application group designated for testing suspicious applications. Then, we applied the following ICC policies to revoke this application group's

capabilities to access user's contacts. The experiments on 9 Gone60 samples demonstrated that TRIPLEMON successfully denied the accesses on the contacts.

```
"ICCPolicy_Gone60": {
    "type"   : "ICC"
    "target": {
        "subject"    : ["GROUP_suspicious"],
        "resource"   : ["com.android.contacts"],
        "action"     : ["ContentProvider", "Activity"]
    },
    "condition"      : ["*"],
    "effect"         : "deny" }
```

5.3.2. *BinderMon vs coarse-grained permissions*

Next, we validated BINDERMON by partially revoking the capabilities covered by a permission called `READ_PHONE_STATE` for privacy purposes. `READ_PHONE_STATE` is a commonly abused permission because it allows applications to call an API `getDeviceID` and read the unique device identifier which could facilitate user tracking. However, simply revoking this permission could break applications because this permission also covers APIs other than `getDeviceID`. To protect user's privacy, we employed the following Binder policies to revoke an application's capability to call `getDeviceID` without affecting the other APIs. In our experiments on 20 randomly selected malware samples that use `getDeviceId`, BinderMon denied every request to `getDeviceID`. We further verified the results by inspecting the files and network traces generated by the malware samples. And we discovered that no information related to device ID was leaked.

```
"BinderPolicy_CapabilityRevoking": {
    "type"   : "BINDER"
    "target": {
        "subject"    : ["GROUP_suspicious"],
        "resource"   : ["iphonesubinfo"],
        "action"     : ["Call"]
    },
    "condition"      : ["cmd=1"],
    "effect"         : "deny" }
```

5.3.3. *OSMon vs root exploits*

In Table 5, we show a list of known root exploits [50], vulnerabilities, and programs attacked by the exploits. We also show the hooks of TRIPLEMON used to prevent corresponding exploits from gaining root privileges. Denying `setuid` is a straightforward countermeasure and can prevent all of exploits from escalating their privileges. We also employed alternative hooks to protect the target of exploits. For example, `Exploid` sends malformed Netlink messages to the kernel via `/proc/sys/kernel/hotplug`. We can mitigate this exploit by revoking its capabilities to use Netlink, and/or use `/proc/sys/kernel`. In addition, most exploits attempt to remount `system` as read-write to retain their root privileges even after the reboot. Thus, the filesystem hooks of TRIPLEMON are helpful to neutralize such attempts.

In our experiments, we tested two exploits, GingerBreak and ZergRush, which have been used by recent malicious applications [11,12]. GingerBreak and ZergRush attack Vold's Netlink socket and local socket, respectively. We defined the following policies to deprive their capabilities to access Vold.

Table 5
Root exploits and countermeasures with TRIPLEMON

Root exploit	Vulnerable program	OSMon hooks
Asroot [8]	kernel	localsocket, setuid
Exploid [7]	init	netlink, setuid, inode
Zimperlich [52]	zygote	task_create, setuid
RAtC [19]	adbd	task_create, setuid
KillingInTheNameOf [9]	ashmem	setuid
Psneuter [9]	ashmem	setuid
GingerBreak [11]	vold	netlink, setuid
ZergRush [12]	libsysutils	localsocket, setuid
Mempodipper [13]	kernel	inode, setuid

```
"OSPolicy_Gingerbreak": {
    "type"   : "OS"
    "target": {
        "subject"    : ["GROUP_suspicious"],
        "resource"   : ["vold"],
        "action"     : ["netlink"]
    },
    "condition"      : ["cmd=netlink_send"],
    "effect"         : "deny" }
```

```
"OSPolicy_ZergRush": {
    "type"   : "OS"
    "target": {
        "subject"    : ["GROUP_suspicious"],
        "resource"   : ["vold"],
        "action"     : ["localsocket"]
    },
    "condition"      : ["cmd=socket_connect"],
    "effect"         : "deny" }
```

We tested the exploits using (1) samples of GingerMaster, and (2) the shell of Android Debugging Bridge (ADB). Our experimental results showed that OSMon can successfully intercept and prevent such exploits. Figure 6 depicts that ZergRush failed because OSMon denied its attempt to crash `vold`.

We further examined a list of known root exploits.[2] We analyzed their exploited vulnerabilities and source code to verify the feasibility of mitigating them with TRIPLEMON. In general, we found that most exploits take advantage of certain system resources that general applications would not write into, such as `/data/data/recovery/log` and `/dev/graphics/fb0`. Indeed, Android applications are expected to access system resources (*e.g.*, device nodes) indirectly via Android APIs. Therefore, OSMon is able to revoke the unnecessary capabilities and prevent such exploits. Note that OSMon does not prevent exploits that use the necessary capabilities of applications. For example, libperf_event exploits the kernel using a crafted system call [14].

[2] https://github.com/droidsec/droidsec.github.io/wiki/Vuln-Exploit-List.

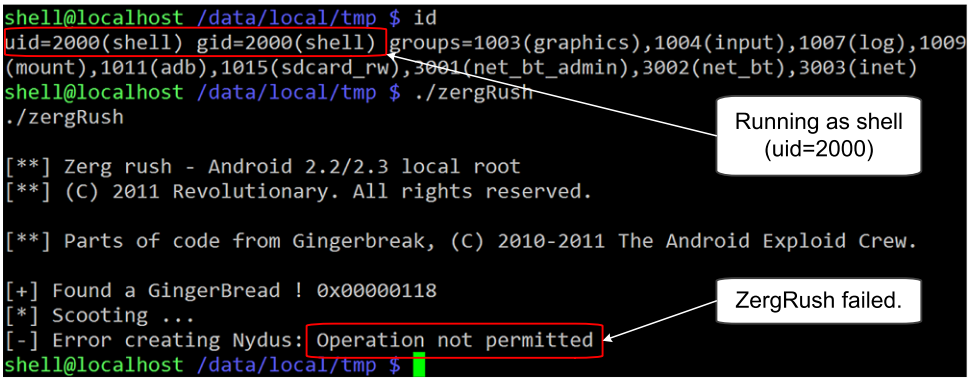

Fig. 6. ZergRush fails to exploit Vold.

5.3.4. Decision reconciliation

Next we evaluated how the decision managers helped coordinate the reference monitors. An Android application may read/write the SMS storage with a two-step method: acquiring a handle to the SMS database via ICC, and then accessing the database file.

We defined a policy set to mediate applications' accesses on SMS. The policy rules in the same policy set protect the same resource (SMS) that corresponds to the objects at multiple layers. As each access involves multiple steps, we define a policy for each step. The decisions made by two policies were aggregated, and the final decision was then made by leveraging the `deny-overrides` strategy. Our experimental results showed that the access was finally denied because the second policy evaluated to deny which overrode the first policy.

```
PolicySet_SMS_ReadWrite: {
    "SMS_ReadWrite_1": {
        "type"   : "ICC",
        "target": {
            "subject"    : ["GROUP_suspicious"],
            "resource"   : ["com.android.providers.telephony"],
            "action"     : ["ContentProvider"]
        },
            "condition"      : ["URI=sms/*"],
            "effect"         : "accept"},
    "SMS_ReadWrite_2": {
        "type"   : "OS",
        "target": {
            "subject"    : ["GROUP_suspicious"],
            "resource"   : ["/data/data/com.android.providers.telephony/mmssms.db
                "],
            "action"     : ["file"]
        },
        "condition"      : ["cmd=dentry_open"],
        "effect"         : "deny"}
}
```

Table 6
Performance overheads compared to its related work on the same device

	μ in ms	σ in ms
ICCMon	0.132	1.060
BinderMon	2.392	4.653
OSMon	0.044	0.036
FlaskDroid	0.337	3.492

Table 7
Memory overhead

File	Original (KB)	TRIPLEMON (KB)	Overhead (KB)
Services.jar	1155.00	1157.82	2.82
LibBinder	1766.54	1782.02	15.48
Kernel (boot.img)	4571.14	4587.52	16.38
Decision manager	N/A	13.71	13.71

5.4. Performance

We evaluated the performance of TRIPLEMON using policies shown in Section 5.3 on a Samsung Galaxy Nexus Device. In addition, we evaluated FlaskDroid's overhead with basic policies used in [4] on the same device. As shown in Table 6 which presents the mean execution time μ and the standard deviation σ for performing a policy check, TripleMon imposes imperceptible runtime overhead. If we break it down, ICCMon incurs less runtime overhead compared with its counterpart in FlaskDroid. BinderMon is a unique component in our design, and its performance is incomparable to FlaskDroid. OSMon, which was implemented as a standard Linux security module, also only introduces negligible overhead. In addition, the average footprint of TRIPLEMON is negligible compared to the sizes of existing components, as shown in Table 7.

6. Discussion

The policy generation process using benign and malicious applications is limited by the fact that dynamic random fuzzy testing fundamentally fails to reveal all possible execution paths. Recent work [27] shows that the coverage can be lower than 40%. Thus, the policy still has a lot of space to improve. For the malware samples used in our evaluation, we observed that their malicious payloads usually executed immediately after the applications started. Therefore, fuzzy testing over a large number of malware samples is helpful for understanding common malicious behaviors to be denied in our policy.

In TRIPLEMON, we manually and statically analyze the Andorid IPC subsystem to identify choke points, *i.e.*, places for inserting authorization hooks. We believe that an automated approach should be explored to comprehensively and systematically identify potential missing choke points. For example, we could introduce dynamic information flow tracking to identify various points that "forward" information flows between different domains. A similar approach has been proposed in [35] to specifically address client–server software. Such approaches could facility the design of reference monitors that can provide better assurance towards complete mediation.

The current implementation of TRIPLEMON is a proof-of-concept prototype so usability analysis and improvement would be another area to be explored. Despite that TRIPLEMON opts for a simple format

of policy scheme with relatively rich expressiveness to ease the burden of policy management, it would be helpful to have user-friendly utilities for creating, maintaining and synchronizing policies. We have implemented a simple web-based management interface for TRIPLEMON. We will continue improving its usability in our future work.

7. Related work

Android security mechanisms have attracted significant attention in recent years. A large number of research projects have been conducted for designing and implementing security extensions on Android to tackle a variety of specific attacks.

FlaskDroid [4] is a two-layer MAC framework that provides flexible and fine-grained mandatory access control on both Android's middleware and kernel layers. It bears the most similarity with our framework. While FlaskDroid and TRIPLEMON both opt for a multi-layer architecture to address the respective semantics of each layer, TRIPLEMON utilizes the peculiarities of Android IPC to be generic and efficient. Specifically, TRIPLEMON's middleware MAC interposes the Binder IPC and ICC channels between applications and system services. This design choice allows TRIPLEMON to enforce system-wide access control polices without modifying system services. FlaskDroid's exemplary implementation provides 12 User Space Object Managers to monitor 40 APIs while TRIPLEMON covers 1,448 public and hidden APIs.

Smalley et al. [44] proposed SEAndroid that extends SELinux as kernel-level MAC and adopts a set of middleware extensions to support middleware MAC. Unlike TRIPLEMON, SEAndroid middleware MAC is only responsible for passing middleware contexts to kernel MAC. The underlying kernel MAC, despite that it has limited semantics of other layers, makes decisions for events occurred at the middleware layer. This situation limits SEAndroid middleware MAC in mitigating corresponding attacks in a fine-grained and accurate manner. In contrast, TRIPLEMON provides reference monitoring at different layers to tackle their respective semantics and employs decision reconciliation to address the possible inconsistencies among them.

XManDroid [2] and TrustDroid [3] both are multi-layer security frameworks that adopt TOMOYO Linux as the underlying kernel MAC and a set of middleware extensions for middlware MAC. Their middleware MAC implementations are tailored to their specific problems: XManDroid [2] attempts to mitigate privilege escalation attacks and TrustDroid [3] establishes an isolated domain for business applications. Along these lines, TRIPLEMON is a generic security framework and can adjust to different threat models and security requirements with user-specified access control policies.

QUIRE [18] enables provenance in Android IPC by propagating verifiable signatures along IPC chains. The signature provides context of the sender application so that a recipient can authenticate the origin of the data they received indirectly. However QUIRE requires that applications must be modified to support QUIRE-style IPC, which is infeasible for most applications whose source code is not available to general users. In contract to QUIRE, TRIPLEMON works with unmodified applications to maintain the compatibility of existing applications.

IPC Inspection [26] is a security framework for tackling permission re-delegation attacks. IPC Inspection reduces the privileges of a recipient application to the intersection of permissions of applications along the IPC chain. However, automatically restricting permissions for collaborative applications is not a decent solution in some cases and may lead to usability loss. Our work employs a policy-based approach and users can specify a group of her trusted applications which can collaborate with each other without restrictions.

TISSA [51] is a policy-driven security extension that protects user's private data. TISSA implements a privacy mode where access to private data can be dynamically and independently controlled. TISSA puts hooks in several privacy-related system services, such as LocationManagerService and TelephonyManagerService. The hooks redirect control flows to a centralized decision maker. Compared to TISSA, TRIPLEMON provides broader coverage of user's data by mediating most IPC channels used by Android applications.

Several recent work addressed fine-grained and context-aware ICC mediation. SAINT [38] is a policy-driven framework that enforces semantically rich policies on ICC at runtime and during installation. Apex [36] provides a similar solution where users can specify runtime constraints for applications. CRePe [6] enables context-aware ICC where environmental constraints such as location and time can be considered for policy enforcement. Although these extensions are not sufficient to cover all existing attacks discussed in Section 2.1, they demonstrate the necessity and value of flexible and fine-grained access control in ICC. Inspired by their work, TRIPLEMON includes a dedicated sub-monitor that addresses the attacks at this layer.

TRIPLEMON requires modifying the Android platform. Although an automated installation kit can reduce the deployment overhead, it cannot entirely eliminate it. Recent research [16,33,43,47] proposed inlined reference monitors by placing hooks inside applications instead of TRIPLEMON's system-centric approach. For example, Aurasium [47] inserts native bootstrapping code into compiled Android applications so as to interpose Libc and mediate Linux system calls. However, such reference monitors are prone to be subverted because they run with the same privileges as the code they are attempting to confine. And Hao et al. [29] demonstrates several potential attacks which may render such reference monitors ineffective or infeasible.

8. Conclusion

In this paper, we have presented the design and implementation of a multi-layer security framework, TRIPLEMON, that provides flexible and fine-grained access control on Android. TRIPLEMON could mediate multiple Android IPC channels (namely, ICC, Binder IPC and Linux IPC) to prevent prominent attacks that could bypass the existing Android security mechanisms. TRIPLEMON monitors and determines the suspicious behaviors of applications that would lead to appropriate policies for mitigating the attacks. Our experiments showed the common behaviors of Android malware in the wild, and demonstrated the effectiveness and practicality of our approach. The performance measurements also showed that our system has produced only a manageable performance overhead.

Acknowledgments

This work was partially supported by the grants from Global Research Laboratory Project through National Research Foundation (NRF-2014K1A1A2043029) and the Center for Cybersecurity and Digital Forensics at Arizona State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the funding agencies.

References

[1] M. Backes, S. Bugiel, S. Gerling and P. von Styp-Rekowsky, Android Security Framework: Extensible multi-layered access control on Android, in: *30th Annual Computer Security Applications Conference (ACSAC'14)*, ACM, 2014.

[2] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi and B. Shastry, Towards taming privilege-escalation attacks on Android, in: *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)*, San Diego, CA, 2012.
[3] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A. Sadeghi and B. Shastry, Practical and lightweight domain isolation on Android, in: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ACM, 2011, pp. 51–62. doi:10.1145/2046614.2046624.
[4] S. Bugiel, S. Heuser and A.-R. Sadeghi, Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies, in: *22nd USENIX Security Symposium (USENIX Security '13)*, USENIX, 2013.
[5] E. Chin, A. Felt, K. Greenwood and D. Wagner, Analyzing inter-application communication in Android, in: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ACM, 2011, pp. 239–252.
[6] M. Conti, V. Nguyen and B. Crispo, CRePE: Context-related policy enforcement for Android, in: *Information Security*, 2011, pp. 331–345. doi:10.1007/978-3-642-18178-8_29.
[7] CVE-2009-1185, Exploid exploit for Android, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1185.
[8] CVE-2009-2692, Asroot exploit for Android, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2692.
[9] CVE-2011-1149, KillingInTheNameOf exploit for Android, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1149.
[10] CVE-2011-1717, Skype stores sensitive user data in files that have weak permissions, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1717.
[11] CVE-2011-1823, GingerBreak exploit for Android, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1823.
[12] CVE-2011-3874, ZergRush exploit for Android, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3874.
[13] CVE-2012-0056, Mempodipper exploit for Android, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0056.
[14] CVE-2013-2094, Libperf_event exploit for Android, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2094.
[15] L. Davi, A. Dmitrienko, A. Sadeghi and M. Winandy, Privilege escalation attacks on Android, in: *Information Security*, 2011, pp. 346–360. doi:10.1007/978-3-642-18178-8_30.
[16] B. Davis, B. Sanders, A. Khodaverdian and H. Chen, I-arm-droid: A rewriting framework for in-app reference monitors for Android applications, in: *Proceedings of the IEEE Mobile Security Technologies*, 2012.
[17] DefCon 18, These are not the permissions you're looking for, 2010, http://goo.gl/sxHyV.
[18] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu and D. Wallach, Quire: Lightweight provenance for smart phone operating systems, in: *20th USENIX Security Symposium*, 2011.
[19] Droid2, http://c-skills.blogspot.com/2010/08/droid2.html.
[20] W. Enck, Defending users against smartphone apps: Techniques and future directions, in: *Information Systems Security*, 2011, pp. 49–70. doi:10.1007/978-3-642-25560-1_3.
[21] W. Enck, M. Ongtang and P. McDaniel, Understanding Android security, *IEEE Security & Privacy* **7**(1) (2009), 50–57. doi:10.1109/MSP.2009.26.
[22] W. Enck, M. Ongtang and P. McDaniel, On lightweight mobile phone application certification, in: *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ACM, 2009, pp. 235–245.
[23] A. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, Android permissions demystified, in: *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM, 2011, pp. 627–638.
[24] A. Felt, M. Finifter, E. Chin, S. Hanna and D. Wagner, A survey of mobile malware in the wild, in: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ACM, 2011, pp. 3–14. doi:10.1145/2046614.2046618.
[25] A. Felt, K. Greenwood and D. Wagner, The effectiveness of application permissions, in: *Proceedings of the USENIX Conference on Web Application Development*, 2011.
[26] A. Felt, H. Wang, A. Moshchuk, S. Hanna and E. Chin, Permission re-delegation: Attacks and defenses, in: *Proceedings of the 20th USENIX Security Symposium*, 2011.
[27] P. Gilbert, B. Chun, L. Cox and J. Jung, Automating privacy testing of smartphone applications, Technical Report CS-2011-02, Duke University, 2011.
[28] L.S. Group, Zero-permission Android applications, http://leviathansecurity.com/blog/archives/17-Zero-Permission-Android-Applications.html.
[29] H. Hao, V. Singh and W. Du, On the effectiveness of api-level access control using bytecode rewriting in Android, in: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ACM, 2013, pp. 25–36.
[30] T. Harada, T. Horie and K. Tanaka, Task oriented management obviates your onus on Linux, in: *Linux Conference*, 2004.
[31] IDC, Smartphones expected to grow 32.7% in 2013 fueled by declining prices and string emerging market demand, according to IDC, June 2013, http://www.idc.com/getdoc.jsp?containerId=prUS24143513.
[32] T. Jaeger, Reference monitor, in: *Encyclopedia of Cryptography and Security*, 2nd edn, 2011, pp. 1038–1040.

[33] J. Jeon, K.K. Micinski, J.A. Vaughan, A. Fogel, N. Reddy, J.S. Foster and T. Millstein, 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*, ACM, 2012, pp. 3–14. doi:10.1145/2381934.2381938.
[34] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo and D. Lin, Access control policy combining: Theory meets practice, in: *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, ACM, 2009, pp. 135–144. doi:10.1145/1542207.1542229.
[35] D. Muthukumaran, T. Jaeger and V. Ganapathy, Leveraing 'choice' in authorization hook placement, in: *19th ACM Conference on Computer and Commumications Security*, 2012.
[36] M. Nauman, S. Khan and X. Zhang, Apex: Extending android permission model and enforcement with user-defined runtime constraints, in: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ACM, 2010, pp. 328–332.
[37] OASIS, Extensible access control markup language (XACML) version 3.0, OASIS standard, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.
[38] M. Ongtang, S. McLaughlin, W. Enck and P. McDaniel, Semantically rich application-centric security in Android, in: *Annual Computer Security Applications Conference, ACSAC'09*, IEEE, 2009, p. 340–349.
[39] M. Ongtang, S. McLaughlin, W. Enck and P. McDaniel, Semantically rich application-centric security in Android, *Security and Communication Networks* **5**(6) (2012), 658–673. doi:10.1002/sec.360.
[40] N. Reddy, J. Jeon, J. Vaughan, T. Millstein and J. Foster, Application-centric security policies on unmodified Android, Technical Report 110017, UCLA Computer Science Department.
[41] Security alert, New rootsmart Android malware utilizes the GingerBreak root exploit, http://www.csc.ncsu.edu/faculty/jiang/RootSmart/.
[42] Security alert, New Android malware – dkfbootkit – moves towards the first android bootkit, http://www.csc.ncsu.edu/faculty/jiang/DKFBootKit/.
[43] P.J. Sheehan, B. Davis and H. Chen, Rewriting an Android app using retroskeleton, in: *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ACM, 2013, pp. 483–484. doi:10.1145/2462456.2465738.
[44] S. Smalley and R. Craig, Security Enhanced (SE) Android: Bringing flexible MAC to Android, in: *NDSS*, 2013.
[45] S. Smalley, C. Vance and W. Salamon, Implementing SELinux as a Linux security module, *NAI Labs Report* **1** (2001), 43.
[46] C. Wright, C. Cowan, S. Smalley, J. Morris and G. Kroah-Hartman, Linux security modules: General security support for the Linux kernel, in: *Proceedings of the 11th USENIX Security Symposium*, Vol. 5, San Francisco, CA, 2002.
[47] R. Xu, H. Saïdi and R. Anderson, Aurasium: Practical policy enforcement for android applications, in: *Proceedings of the 21st USENIX Security Symposium*, 2012.
[48] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C.A. Gunter and K. Nahrstedt, Identity, location, disease and more: Inferring your secrets from Android public resources, in: *Proceedings of the ACM Conference on Computer and Communications Security*, 2013.
[49] X. Zhou, Y. Lee, N. Zhang, M. Naveed and X. Wang, The peril of fragmentation: Security hazards in Android device driver customizations, in: *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE, 2014.
[50] Y. Zhou and X. Jiang, Dissecting Android malware: Characterization and evolution, in: *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*, San Francisco, CA, 2012.
[51] Y. Zhou, X. Zhang, X. Jiang and V. Freeh, Taming information-stealing smartphone applications (on Android), in: *Trust and Trustworthy Computing*, 2011, pp. 93–107. doi:10.1007/978-3-642-21599-5_7.
[52] Zimperlich sources, http://c-skills.blogspot.com/2011/02/zimperlich-sources.html.