Pervasive and Mobile Computing 30 (2016) 113–131

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Android vs. SEAndroid: An empirical assessment☆

Alessio Merlo[a,*], Gabriele Costa[a], Luca Verderame[a], Alessandro Armando[a,b]

[a] DIBRIS, Università degli Studi di Genova, Italy
[b] Security & Trust Unit, FBK-irst, Trento, Italy

ARTICLE INFO

Article history:
Received 6 August 2015
Received in revised form 18 November 2015
Accepted 7 January 2016
Available online 21 January 2016

Keywords:
Android OS
Android security
Android Security Framework
SEAndroid
SELinux MAC

ABSTRACT

Android has a layered architecture that allows applications to leverage services provided by the underlying Linux kernel. However, Android does not prevent applications from directly triggering the kernel functionalities through system call invocations. As recently shown in the literature, this feature can be abused by malicious applications and thus lead to undesirable effects. The adoption of SEAndroid in the latest Android distributions may mitigate the problem. Yet, the effectiveness of SEAndroid to counter these threats is still to be ascertained. In this paper we present an empirical evaluation of the effectiveness of SEAndroid in detecting malicious interplays targeted to the underlying Linux kernel. This is done by extensively profiling the behavior of honest and malicious applications both in standard Android and SEAndroid-enabled distributions. Our analysis indicates that SEAndroid does not prevent direct, possibly malicious, interactions between applications and the Linux kernel, thus showing how it can be circumvented by suitably-crafted system calls. Therefore, we propose a runtime monitoring enforcement module (called *Kernel Call Controller*) which is compatible both with Android and SEAndroid and is able to enforce security policies on kernel call invocations. We experimentally assess both the efficacy and the performance of KCC on actual devices.

1. Introduction

Android consists of a Java stack built on top of a native Linux kernel. Services and functionalities are achieved through the interplay of components residing at different layers of the operating system. The Android Security Framework (ASF) consists of a number of cross-layer security solutions combining basic Linux security mechanisms (e.g. Discretionary Access Control), the app isolation offered by the Java Virtual Machine execution environment and Android-specific mechanisms (e.g. the Android permission system).

The security offered by the ASF has been recently challenged by the discovery of a number of vulnerabilities involving all layers of the Android stack (see, e.g., [1–3]). The analysis of these interplay-related vulnerabilities indicates that

- the security mechanisms of the Android stack (both Java native and Android-specific) are not completely integrated with those in the Linux kernel, thereby allowing insecure interplay among layers;
- malicious and unprivileged Android applications can directly interact with the underlying Linux kernel, thereby by-passing the controls performed by the ASF.

☆ This work has been partially funded by the Italian PRIN project *Security Horizons* (no. 2010XSEMLC).
* Corresponding author.
E-mail addresses: alessio.merlo@unige.it (A. Merlo), gabriele.costa@unige.it (G. Costa), luca.verderame@unige.it (L. Verderame), armando@fbk.eu (A. Armando).

http://dx.doi.org/10.1016/j.pmcj.2016.01.006
1574-1192/

In [1] and [4] we reported a vulnerability in cross-layer interaction that allowed to mount a fork bomb attack on all Android distributions up the release of a patch for 4.0.3 version, in [5] we have carried out an empirical evaluation aimed at determining to which extent such a lack of control in the ASF may allow applications to maliciously trigger the Linux kernel functionality by means of properly-forged invocations.

The introduction of the *Security Enhancements for Android* project (SEAndroid) [6] in the Android Open Source Project (AOSP) since version 4.4.3 calls for a reconsideration of the results reported in [5]. In fact, according to the official documentation, the SEAndroid project aims at enabling the use of SELinux in Android *"in order to limit the damage that can be done by flawed or malicious applications and in order to enforce separation guarantees between applications"* [7] and it should therefore mitigate the aforementioned problems. It must also be observed that SEAndroid suffers from a number of limitations. For instance, the development of a *"good policy"* and the trustworthiness of the ASF are *"crucial to the effectiveness of SE Android"* [7].

To illustrate, let us consider the Zygote vulnerability reported in [1]. Such a vulnerability allows a malicious application to force the Linux kernel to fork an unbounded number of processes thereby making the device totally unresponsive. In this case, the problem is due to the fact that the ASF is not able to discriminate between a legal interplay (carried out by trusted Android services) and an insecure one (executed by applications), thereby permitting the direct invocation of a critical kernel functionality (i.e. the fork operation) by any application. This is basically due to a lack of control on Linux system calls involved in the launch of applications. Although SEAndroid is able to address the Zygote vulnerability (by limiting the usage of socket devices by applications), it *"cannot in general mitigate kernel vulnerabilities"* [7].

This paper extends the methodology proposed in [5] in a number of ways:

1. An experimental setup for SEAndroid is discussed. The experimental setup is aimed at (i) systematically capturing invocations to the Linux kernel from different layers in the Android stack, and (ii) replicating the invocations through a properly-crafted application.
2. A new empirical assessment, involving both recent Android and SEAndroid builds, has been carried out and a comparison between the detection capabilities of Android and SEAndroid is discussed.
3. An interplay that allows a malicious application to circumvent SEAndroid is shown. This witnesses the limited control exercised by SEAndroid on the interactions between applications and the Linux kernel.
4. A policy enforcement module, called *Kernel Call Controller* (KCC for short), that provides both Android and SEAndroid with the possibility to recognize and limit the direct interaction between the Android stack and the Linux kernel is presented.
5. An empirically-inferred KCC policy for filtering kernel call invocations according to (i) the identity of the caller and (ii) the number of repeated invocations, is discussed.
6. KCC performance and reliability is evaluated and discussed using the sample policy.

Structure of the paper. Section 2 provides a general introduction to the cross-layer architecture of Android. Section 3 discusses the peculiarities and limitations of the ASF whereas Section 4 introduces SEAndroid and discusses its security features. Section 5 describes the implementation and setup of our assessment environment for both Android and SEAndroid. Section 6 is devoted to describing the experimental setup and results. Section 7 discusses the effectiveness of the security countermeasures provided by the ASF and SEAndroid against two malicious applications. Section 8 introduces the *Kernel Call Controller* module, the language used for defining enforcement policies and its empirical assessment using a empirically-inferred policy. Section 9 discusses some related work while Section 10 concludes the paper with some final remarks and future directions.

2. Android in a nutshell

The Android architecture consists of 5 layers. The Linux kernel lives in the bottom layer (henceforth the *Linux kernel*). The remaining four layers are Android-specific and we therefore collectively call them *the Android stack*:

Application Layer *(*A*).* Applications are at the top of the stack and comprise both user and system applications that have been installed and executed on the device. Android applications are built as a set of independent execution modules called *components*. The reader can refer to [4] for further details regarding application components and their interactions.

Application Framework Layer *(*AF*).* The Application Framework provides the main services of the platform that are exposed to applications as a set of APIs. This layer provides the System Server, that is a process containing the Android core components[1] for managing the device and for interacting with the underlying Linux drivers.

Android Runtime Layer *(*AR*).* This layer consists of the Dalvik Virtual Machine (Dalvik VM, for short), i.e. the Android runtime core component that executes application files built in the Dalvik Executable format (.dex). The Dalvik VM is specifically optimized for efficient concurrent execution of virtual machines in a resource constrained environment.

[1] http://events.linuxfoundation.org/slides/2011/abs/abs2011_yaghmour_internals.pdf for a comprehensive list of such service components.

Libraries Layer *(*L*).* The Libraries layer contains a set of C/C++ libraries that support the direct invocation of basic kernel functionalities. They are widely used by services in the AF layer to interact with the Linux kernel and to access data stored on the device. An Example of library is *Bionic libc*, a derivation of the standard C implementation for a mobile environment. In Android, this library is included in each Linux process.

Kernel layer *(*K*).* Android currently relies on the Linux kernel version 3.x for core system functionalities. These functionalities include (i) the access to physical resource (i.e. device peripherals, memory, file system) and (ii) the Inter-Process Communication (IPC). Device peripherals (e.g. GPS antenna, Bluetooth/Wireless/3G modules, camera, accelerometer) are accessed through Linux drivers installed as kernel modules. Triggering peripheral drivers, as well as accessing file system and memory are achieved by means of *system calls* (e.g. `open`, `read` and `write` for files management). IPC may be carried out through the use of the Binder driver or by reading from/writing to native Unix Domain Sockets. The Binder driver is activated through *binder* calls (i.e. `ioctl`), while sockets are accessed through *socket calls* (e.g. `connect`, `bind`, `sendmsg`).

2.1. Notes on the interplay in android

Operations in Android are carried out through interactions among layers. Such interactions constitute the interplay of Android and are implemented through 6 kinds of calls (namely, *function*, *dynamic load*, *jni*, *system*, *binder*, and *socket* calls) involving distinct subsets of layers and libraries (see [4] for details on calls). *System*, *binder*, and *socket* calls allow to trigger directly the Linux kernel functionalities. Hereafter, we refer to these kinds of calls as *kernel calls*.

Android provides OS functionalities to applications by means of combinations of calls. For instance, the launch of a new application in Android is normally provided by the following interplay:

1. a requesting application (i.e. the home screen of the device) (A layer) executes a *binder call* to the Activity Manager Service (AMS) (AF layer) to start the launching process.
2. The AMS checks the permissions of the requesting application and, in case they are sufficient, executes a set of *socket calls* (i.e. `connect`, `sendmsg`, `listen`) to the Zygote socket (K layer) for writing down a command aimed at requesting the launch of a new application. The command contains information related to the application to launch.
3. The controlling process of the Zygote socket (i.e. the Zygote process) parses the command and invokes a *JNI call* to load a proper library function (L layer) for accessing kernel functionalities.
4. The invoked function directly executes a `fork` *system call* at K layer, building a new Linux process that will host the launching application. If something goes wrong, the command provided by the AMS to the Zygote socket forces the kernel to destroy the created process, otherwise a new Dalvik VM with the code of the launching application is bound to the process and the execution is started.

Interplay in Android is poorly documented and not standardized. As a matter of fact, the interplay of only a few operations is discussed in the official literature [8] and the interplay related to the launch of a new application is not documented. The above description (borrowed from [1]) has been inferred by systematically analyzing Android source code. Moreover, the lack of documentation and standardization implies that the same functionalities could be potentially carried out through different interplay, some of which may lead to security flaws. To prevent this, the Android Security Framework discriminates whether an interplay is secure or not, according to the permissions of applications and the basic Android security policy. We introduce in the following the basis of the ASF and then we reason about its limitations related to the analysis of the interplay.

3. The android security framework

The Android Security Framework (ASF) provides a cross-layer security solution (i.e. sandboxing) built by combining native per-layer security mechanisms. Each layer in the Android stack (except the Libraries layer) comes with its own security mechanisms:

- **Application layer (Android Permissions)**. Each application comes with a file named `AndroidManifest.xml` that contains the description of the components composing the application as well as the permissions that the application may require during execution. The user is asked to grant all the permissions specified in the manifest to properly install and execute the application.
- **Application Framework (Permission Enforcement)**. At runtime, services at this layer enforce the permissions specified in the manifest and granted by the user during installation.
- **Runtime (VM Isolation)**. Each application is executed in a separate Dalvik VM machine. This ensures isolation among applications.
- **Linux (Access Control)**. As in any Linux kernel, resources are mapped into files (e.g. sockets, drivers). The Linux Discretionary Access Control (DAC) model associates each file with an owner and a group. Then, DAC model allows the owner to assign an access control list (i.e. read, write, and/or execute) on each file to the owner itself (UID), the owner's group (GID) and other users.

Sandboxing is a cross-layer solution adopted in Android to provide strong isolation among applications. In detail, Android achieves sandboxing of the applications by binding each Android application to a separate user at Linux level, thereby combining the native separation due to the execution of applications on different Dalvik VMs with the isolation provided by native Linux access control.

Once an application is installed on the device (i.e. the user accepts all required permissions in the `AndroidManifest.xml` file), a new user at the Linux layer is created and the corresponding user id (UID) is bound to the installed application. Once the application is launched, a new process, with such UID, and a novel Dalvik VM are created in order to execute the application.

This solution forces any non privileged UID to have at most one process running (i.e. the one containing the running application). Rarely, more than one active process for the same UID can be allowed if explicitly requested in the `AndroidManifest.xml`. However, the maximum number of active processes is upper-bounded by the number of components composing the application.

At runtime, sandboxing and other per-layer security mechanisms are expected to avoid illegal interplay. For instance, if an application tries to invoke a `kill` system call on the process hosting another application, the sandboxing is violated (i.e. a Linux user tries to kill a process belonging to another user) and the system call is blocked.

Example 1. Consider the file management operations in an Android environment.

An application A with package name `com.example.a`, at installation time, is assigned to a home directory (i.e `/data/data/com.example.a`) and a unique UID_A and GID_A. Each file inside the home directory is owned by UID_A and assigned to permissions `rwxrwx---`.

Informally, this means that the file is fully accessible (`rwx`) to A and other members of its group, while being none accessible to anyone else. □

3.1. Security considerations on the android security framework

Some Android vulnerabilities indicate that the integration of the Android-specific security mechanisms with those provided by Linux may suffer from unknown security flaws.

For instance, as shown in [1], in the launching flow presented in Section 2.1 an application can invoke *socket calls* directly and send an ad-hoc command through the Zygote socket, thereby by-passing the Activity Manager Service. The ASF identifies such interplay as legal, since it is not able to notice that an application is invoking calls targeted to the Linux kernel instead of a trusted service in the Application Framework layer. Furthermore, such an interplay can be repeated an arbitrary number of times by the malicious application until the device becomes totally unresponsive.

Other vulnerabilities suggest that security breaches can be hidden in the communications among applications. For instance, single messages exchanged among applications (by means of *binder calls*) can be individually compliant with the applications permissions and hence permitted by the ASF; however, some maliciously crafted interplay can lead to undetected privilege escalation [2] or attacks on legal applications [3].

Previous work, i.e. [5], argues that such problems are basically due to a lack of control on:

- the identification of the caller for direct kernel invocations (i.e. *binder*, *socket* and *system calls*) that may allow malicious applications to operate undetected on the kernel instead of legal Android services;
- the monitoring of cross-layer interplay, that may allow malicious interactions if single calls do no violate any system policy or the sandboxing;
- the identification of repeated interplay, that can make the OS weak against Denial-of-Service attacks.

Previous considerations were substantiated by means of an empirical evaluation of the ASF. However, from version 4.4.3 the Android Open Source Project includes a security enhancement called *SEAndroid* [7] aimed at enhancing the security provided by the native ASF. To this aim, in the following the security features provided by SEAndroid are introduced and an empirical assessment for evaluating the actual security granted by SEAndroid is carried out.

4. SEAndroid

SEAndroid refers to *Security Enhancements for Android*, a solution that has been released at the beginning of 2012 by the National Security Agency and made available in the Android official branch from version 4.4.3.[2] SEAndroid extends the native Android security model with SELinux [9] Mandatory Access Control (*SELinux MAC*). Unlike DAC, where subjects have the possibility to change the access control policy of the resources they own, MAC enforces a centralized access control policy that subjects cannot modify. In addition to the SELinux enhancements to Android, SEAndroid includes a set of extensions to the Android middleware (i.e. the Android Stack) that provides different forms of mandatory restrictions over the Android permissions and Inter-Component Communication (ICC) model. These extensions are collectively called *Middleware MAC* or *MMAC*.

2 http://thehackernews.com/2012/01/security-enhanced-se-android-released.html.

Table 1
Excerpt of SELinux MAC ACM corresponding to the fragment of Example 2.

Subject \ Object	...	app_data_files:dir	...
⋮	⋮	...	⋮
appdomain	⋮	create_dir_perms	⋮
⋮	⋮	...	⋮

4.1. SELinux MAC

SELinux is a MAC solution for the Linux kernel, implemented as a Linux Security Module. The Linux Security Module (LSM) [10] is a framework that supports the inclusion of access control security modules in Linux kernels besides the standard DAC. More in detail, the LSM framework mediates the access to all security-critical operations (e.g. system call execution, inode access, ...) and provides an API interface that triggers an access control policy, e.g. the MAC policy introduced by SEAndroid. To ensure compatibility with existing applications, the LSM enforcement is executed in conjunction with the standard Linux DAC model. This means that, in the SELinux MAC case, a security-critical operation is allowed if and only if it satisfies both the DAC and the MAC policy.

The SELinux MAC model is based on three main concepts: subjects, objects, and actions. Given this model, any action performed by a subject (e.g. a process) on any object (e.g. a file or a socket) is checked against a set of authorization rules (i.e. a security policy) to determine whether the action is allowed or not. Both subjects and objects are labeled with a set of security attributes, called *security contexts*, that are involved in the SELinux decisional process.

In short, a SELinux MAC security policy defines an access control matrix (ACM) on sets of subjects, objects and actions. To clarify this point, the following example is introduced.

Example 2. Let consider again Example 1 and the following rule, extracted from `app.te`[3] of the native SEAndroid security policy.

```
# App sandbox file accesses.
allow appdomain app_data_file:dir create_dir_perms;
```

Informally, the meaning of the rule is that all the applications belonging to `appdomain` can access to the directories contained in `/data/data` (`app_data_files:dir`) through the actions contained in `create_dir_perms`. In terms of MAC, this amounts to say that the SELinux ACM is structured as shown in Table 1.

Assuming no other rule applies, application $A \in$ `appdomain` can access to directory $D \in$ `app_data_files:dir` through action a if and only if $a \in$ `create_dir_perms`. □

From an architectural point of view, the SELinux MAC enforcement is performed by three main components: the Object Manager (OM), the Access Vector Cache (AVC) and the Security Server (SS), as shown in Fig. 1. An Object Manager mediates security-relevant operations and enforces the decisions provided by the SS to the subset of resources it manages.

Furthermore, decisions provided by the SS are cached in the AVC in order to minimize the performance overhead of SEAndroid. More in detail, when a subject queries the corresponding OM to perform an action on an object, e.g. a process tries to read a file, as depicted in step 1 of Fig. 1, the OM queries the AVC to check whether the action is allowed (step 2). If the AVC contains a cached security decision for the request, then it directly answers back to the OM. If this is not the case, the AVC queries (step 3) the SS for a response and return the answer back to the OM. The SS evaluates incoming access control requests querying the SELinux MAC policy (step 4). Finally, the OM grants/denies the action according to the security response (step 5).

4.2. Middleware MAC

The Middleware MAC in SEAndroid allows to enforce SELinux MAC policy on the Android models and data structures (i.e. permission check, application installation and intent propagation).

In principle, the mapping of SELinux MAC access rules, based on low-level entities like processes or files, to high level mechanisms, like permission enforcement or Intent IPC, would basically require to re-write many of the native Android mechanisms [7]. This implies both a huge customization of the Android stack and an extension of the SELinux MAC policy to native Android concepts like intents or permissions.

[3] https://android.googlesource.com/platform/external/sepolicy/+/android-4.4_r1/app.te.

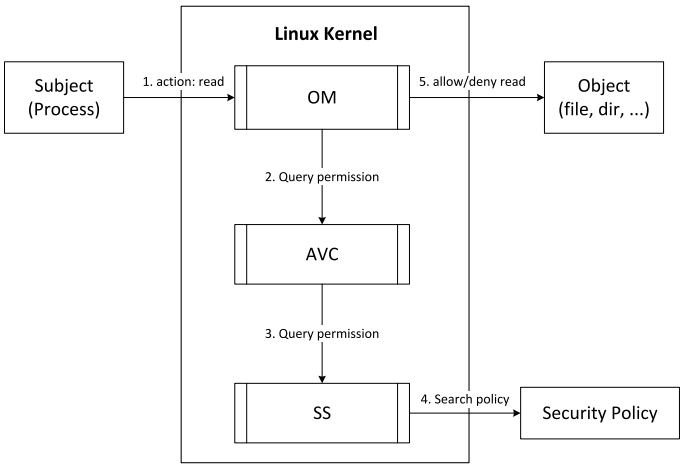

Fig. 1. SELinux MAC architecture.

In order to limit the customization of the Android stack, SEAndroid implements the Middleware MAC (MMAC). The MMAC acts as a high-level access control model with a distinct policy (MMAC policy) and is separated from the SELinux MAC that lives at the kernel level. This choice ensures that a modification in the middleware architecture or policy does not affect the underlying SELinux MAC policy which is kept small and fixed. Since a technical discussion on those changes is out of the scope of this paper, we refer interested readers to [7].

The MMAC original implementation, according to [11], includes:

- **Install-time MAC.** It restricts the installation of applications, based on the MMAC policy located in `mac_permissions.xml`. The policy grants the installation of applications according to their certificate and package name. Applications whose certificate and package name are not explicitly allowed by the policy are rejected, independently from the decision of the user.
- **Permission Revocation.** It allows to revoke permissions at runtime by enhancing the default Android permission check with a policy-based verification. The additional policy verification overrules – and in case may contradict – the default permission check. However, this approach can cause unexpected application behaviors (e.g. crashes, unexpected error handling, ...), since developers rely on the fact that installed application can exploit all permission granted at install time. For these reasons, the SEAndroid development team is currently working on a enhancement called Enterprise Operations (EOPs) which aims at providing support for enterprise control over runtime application operations.[4]
- **Intent MAC.** It protects the delivery of *intents* to Activities, Broadcast Receivers and Services through a white-listing enforcement mechanism. The white-listing is based on attributes of the intent objects (e.g., the value of the action string) and the security type assigned to both the intent sender and the destination applications.

4.3. Security considerations on SEAndroid

SEAndroid architecture aims at enhancing the security of Android distribution, as extensively discussed by authors in [7]. For example, the SELinux MAC prevents privilege escalation and unauthorized data sharing by applications through a rigorous security context labeling and enforcement. Moreover, the Install-time MAC prevents users from installing untrustworthy applications with a potentially dangerous set of permissions.

SEAndroid enforces its security mechanism at low level (i.e. on processes and object), making it hardly applicable at higher abstraction layers. To this aim, SEAndroid extends its security features to the Android upper layers through the Middleware MAC. Unfortunately, MMAC is restricted to application permissions (i.e. Permission Revocation and Install-time MAC) and applications interactions (i.e. Intent MAC) and do not explicitly deal with both the interplay among layers and the kernel call invocations.

An example of interplay among layers is the launch of a new application, described in Section 2.1 and analyzed in Section 3.1. To solve this issue, SEAndroid (i) extends the Zygote implementation with the capability of setting security contexts of processes it forks and (ii) includes specific rules in its policy to restrict access to the Zygote socket.

Although such modifications prevent malicious application, like Zimperlich [12], from gaining the root credentials as well as the *fork bomb* described in [1], this solution is related to a specific and well-known Android vulnerability. Indeed, as stated in [7], the current implementation of SEAndroid may reveal to be rather weak against unknown kernel level vulnerabilities and it does not provide any feature allowing to explicitly verify the interplay among different layers.

[4] See http://seandroid.bitbucket.org/EnterpriseOperationsEOps.html for further details.

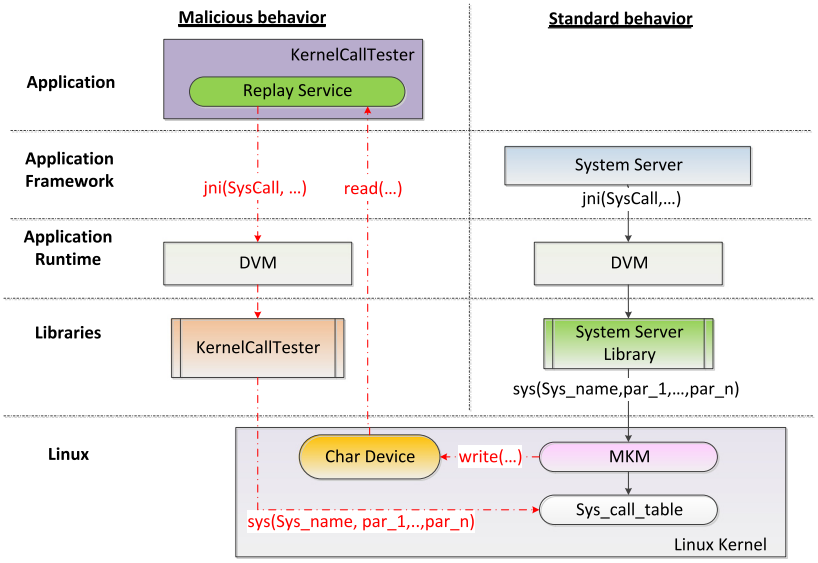

Fig. 2. Interaction between the MKM and the KernelCallTester application.

To ascertain the range and the impact of such limitation on Android security, we carry out an extensive empirical assessment of both native ASF and SEAndroid on cross-layer interplay and kernel call handling. More specifically, the assessment is aimed at evaluating to which extent the native SEAndroid policy – as well as the supported types and security contexts – actually improve the ASF functionality in detecting and limiting malicious direct interplay between applications and the Linux kernel.

5. Assessing android and SEAndroid on kernel calls

The Zygote vulnerability is basically due to a lack of control on the identity of the components invoking a *socket call* targeted to the Kernel layer that is normally expected to be executed by trusted services in the AF layer. Although this vulnerability has been patched since Android version 4.0.3, as discussed in [1], and SEAndroid has modified the Zygote in order to obtain a finer-grained discrimination between applications that are allowed to issue commands on its socket interface, both the original ASF and its SEAndroid-based enhancement may lack in recognizing other malicious interplay between applications and the Linux kernel. More specifically, other kernel calls may be directly executed from applications, thereby bypassing most of the security filtering provided by the ASF, the MMAC and the SELinux MAC.

To this aim, in this section we discuss an experimental setup allowing to (i) systematically identify kernel calls that are normally executed by the trusted services in the AF layer, and (ii) experimentally assess whether the same kernel calls can be replicated by a malicious application directly to the Linux kernel (i.e. bypassing security checks).

The former step is necessary basically due to a very limited documentation on cross-layer interplay in both Android and SEAndroid. More specifically, it is impossible to rely exclusively on current technical and research literature to exhaustively establish which kernel calls are invoked by trusted AF services. To this aim, static analysis techniques could help (they are widely adopted to retrieve models of Android applications from the Dalvik code). However, due to the complexity and the size of the OS source code, static analysis can be complex and cumbersome. Hence, we opted for empirically profiling trusted AF services in terms of the kernel calls they invoke during normal device usage.

Regarding the latter step, our experimental setup is based on an application that tries to replicate the same kernel calls invoked by the trusted AF services. In this way, the experiment empirically assesses to which extent the ASF and the SEAndroid MAC solutions (both at Android stack and the Kernel layer) are able to discriminate and intervene.

5.1. Experimental setup

We set up the experiment into two phases: we implemented (i) a *Monitoring Kernel Module* (MKM) able to intercept kernel calls invoked by the whole *Android stack* and (ii) a tester application (*KernelCallTester*)[5] that is able to replicate the calls intercepted by the MKM. Then, we set up the MKM to intercept kernel calls executed by trusted services in the AF layer and the *KernelCallTester* application to reproduce each kernel call as soon as it has been intercepted (see Fig. 2).

[5] Both MKM and KernelCallTester are publicly available at: https://github.com/ai-lab/SEAndroid-Assessment.

The MKM is a kernel module that customizes the way in which kernel calls are invoked. Upon installation, the MKM retrieves the kernel call prototypes from `systemcalls.h` and the kernel calls numbers from `unistd.h`. Then, it modifies each entry in the `sys_call_table` structure, which contains the kernel calls routines. In details, the MKM substitutes each routine in the table with a customized one. Each customized routine gets the calling thread name and process PID (using the Linux macro `current`) as well as the optional parameters passed to the call and, then, it executes the normal routine. At runtime, the MKM creates a *char device* in order to store the intercepted kernel calls and the corresponding parameters. Each time the MKM intercepts a kernel call, the custom routine writes the data intercepted on the char device.

KernelCallTester has been designed to replicate kernel calls invocations intercepted by the MKM and stored on the char device. For each kernel call invocation, *KernelCallTester* tries to replicate it a random number of times. *KernelCallTester* is composed by a broadcast receiver, an Android service (called `Replay Service`), running on the device in background, and a C++ pre-compiled library. The broadcast receiver is in charge of executing the `Replay Service` once the device completes the boot. The `Replay Service` connects to the char device (located in `/dev/logger`) created by the MKM and parses data contained on it. For each invocation received, the `Replay Service`, by means of a *jni* call to the *KernelCallTester* library, starts replicating the corresponding kernel call with the same parameters as the original one. Depending on the kind of kernel calls, the `Replay Service` may also execute other ad-hoc calls. For instance, if a `read` call is invoked on a file, the `Replay Service` tries to execute an `open` on the same file before invoking the `read`. The `Replay Service` keeps track in proper log files of the success/failure of each replication attempt. Besides, error messages for failed invocations are stored.

5.2. Deploying and configuring MKM and KernelCallTester

We deployed the MKM and the *KernelCallTester* into three Android builds, namely v. 2.3.7 (API 10) and v. 4.1.3 (API 16), as representative distributions without SEAndroid (respectively for entry-level and top-notch smartphones), and into v. 4.4.4 (API 19), which is SEAndroid-enabled.

Since the ability to load generic modules is natively disabled in the Linux kernel deployed in Android, we enabled such feature by recompiling the kernel for a generic ARM architecture. Such modification does not alter any kernel functionality, thus the behavior of the recompiled kernel is equivalent to the original one. Moreover, we have developed a custom `rc.module` script, executed as a service in the `init.rc`, which installs the MKM automatically at startup.

Since the SEAndroid policy restricts the usage of char devices by untrusted applications, we had slightly modified the SEAndroid policy to grant *KernelCallTester* the ability to connect to the char device provided by the MKM. In particular, we added a rule in `untrusted_app.te` SEAndroid policy file:

```
allow untrusted_app device:chr_file {open read write};
```

This modification allows an untrusted application to open and modify (read/write) a character device. Since this change violates an explicit `neverallow` rule, we also need to remove it from `domain.te`:

```
-- neverallow { domain -unconfineddomain -ueventd } device:chr_file  { open
    read write };
```

We configured the MKM to intercept kernel calls executed by trusted services and to keep track of a subset of the most representative kernel calls,[6] including core system calls (e.g. for I/O and process management: `open`, `close`, `read`, `write`, `lseek`, `mkdir`, `rmdir`, `exit_group`, `exit`, `getpid`, `gettid`, `kill`, `lstat64`, `prctl`, `setuid`, `setgid`, `waitid`, `shutdown`, `gettuid`, `geteuid`, `getgid`, `mount`, `umount`), socket calls (`bind`, `connect`, `sendmsg`, `sendto`, `socket`, `recvfrom`, `recvmsg`, `listen`) and binder calls (which rely on the `ioctl` system call).

This selection covers a wide range of Android security relevant operations, like file management, Internet connection, IPC communication and launch of new applications. Since Kernel modules normally lead to computational overhead, we have chosen to reduce the set of monitored kernel calls to limit the impact of both the MKM and *KernelCallTester* on the performance of the testing devices. More specifically, we experienced that profiling each kernel call leads to a significant performance degradation that may affect the normal usage of devices; hence, in order to grant the usability of devices during the testing phase, we reduced the number of profiled kernel calls to the most security-relevant ones. In this respect, in Section 8.5 we experimentally assess the impact that capturing and analyzing kernel call invocations at runtime may have on the performance of the device.

6. Testing and experimental results

We installed the three customized Android builds presented in Section 5.2 into a set of thirteen smartphones. We deployed the Android build v. 2.3.7 to five smartphones (i.e. HTC Desire HD, HTC Wildfire, LG P500, Samsung GT-I5500,

[6] https://github.com/android/platform_bionic/blob/master/libc/SYSCALLS.TXT.

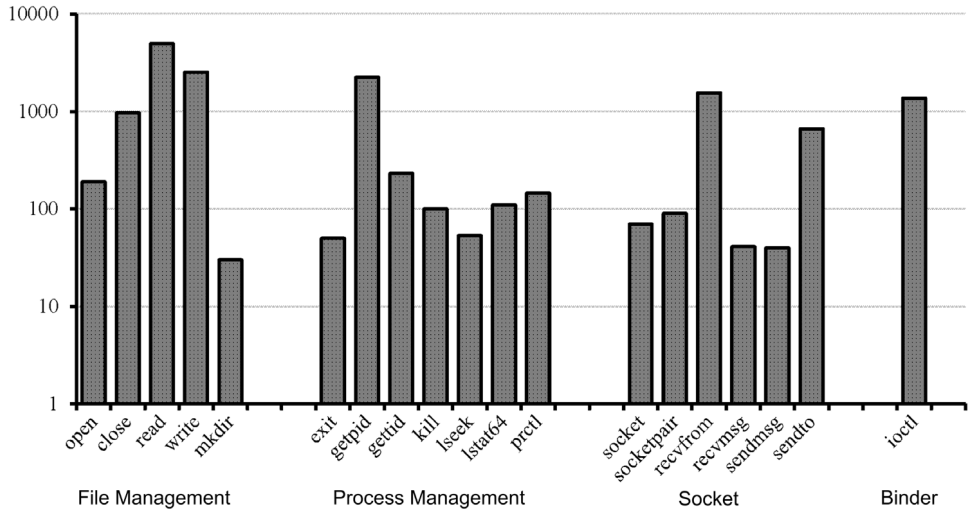

Fig. 3. Excerpt of about 15K Kernel call executed the *AF* on a single device (logscale).

Table 2
Kernel calls invoked by services in the AF layer in Android v. 2.3.7 and v. 4.1.3.

AF service	Kernel calls
Alarm manager	`getpid`, `ioctl`, `open`
Activity Manager	`close`, `getpid`, `gettid`, `ioctl`, `lseek`, `mkdir`, `open`, `prctl`, `read`, `write`
Audio Service	–
BatteryStats	`close`, `exit`, `gettid`, `open`
GpsLocationProvider	`getpid`, `ioctl`
Location Manager Service	`getpid`, `ioctl`, `lseek`, `open`, `read`
Package Manager	`close`, `getpid`, `gettid`, `ioctl`, `lstat64`, `open`, `sendmsg`, `write`
Power Manager Service	`getpid`, `ioctl`, `open`, `read`, `write`
ServerThread	`close`, `connect`, `getpid`, `gettid`, `ioctl`, `lseek`, `lstat64`, `open`, `prctl`, `read`, `recvmsg`, `sendmsg`, `sendto`, `socket`, `write`
ThrottleService	`close`, `exit_group`, `getpid`, `gettid`, `ioctl`, `open`, `prctl`, `read`, `sendmsg`, `write`
VoldConnector	`getpid`, `gettid`, `ioctl`, `open`, `recvmsg`, `write`
Window Manager	`close`, `getpid`, `gettid`, `ioctl`, `open`, `read`, `write`

Samsung GT-S7500), the Android build v. 4.1.3 to other five smartphones (i.e. Galaxy Nexus, Samsung Galaxy Nexus S, Motorola Droid RAZR MAXX, Galaxy Tab 7.1, Sony Xperia T) and SEAndroid v. 4.4.4 to two Nexus 4 and a Nexus 5. Then, we delivered the smartphones to a heterogeneous set of users (i.e. university students, teenagers, professors and clerks) for normal use for two weeks. Users have been left free to install and use every kind of application.

During the testing period the execution of applications had forced services in the AF layer to invoke kernel calls to provide functionalities to applications. By analyzing the MKM logs from the testing smartphones, which reported many thousands of kernel call invocations, we were able to relate services in the AF layer with the kernel calls they invoke. Table 2 summarizes results for Android v. 2.3.7 and v. 4.1.3, while Table 3 shows data related to SEAndroid (v. 4.4.4). The *AF* structure slightly differs between native Android distributions (i.e. v. 2.3.7 and v. 4.1.3) and SEAndroid ones (v. 4.4.4). For instance, SEAndroid *AF* includes a novel service called Network Policy Management Service which enforces low-level network policy rules that are stored on the device and accessed using file management system calls. However, we noticed that such difference does not affect the kind of kernel calls used by the *AF*.

Fig. 3 shows an excerpt of about 15K kernel calls performed by *AF* services in a single testing device, i.e. a Nexus 4 equipped with SEAndroid.

During experimentation, 28 out of 33 (85%) of the kernel call types intercepted by the MKM have been successfully replicated by the *KernelCallTester* both on Android v. 2.3.7 and v. 4.1.3. Only 5 call types (15%) failed due to Linux permissions errors or wrong parameters. Furthermore, experimental results show that 24 out of 33 (75%) of kernel call types can be successfully reproduced also on devices equipped with SEAndroid. More in detail:

System calls. System calls reproduced by *KernelCallTester* can be divided into file management and process management system calls.

- **File management system calls.** The MKM intercepts system calls related to files management as well as the parameters used (e.g. stored data and absolute path of the file opened). Since files are accessed in Linux by means of *file descriptors*, whenever a file system call occurs, *KernelCallTester* tries to reopen the targeted files,

Table 3
Kernel calls invoked by services in the AF layer in SEAndroid.

AF service	Kernel calls
Alarm manager	`getpid`, `ioctl`, `open`, `close`
Activity manager	`close`, `getpid`, `gettid`, `ioctl`, `lseek`, `mkdir`, `open`, `prctl`, `read`, `write`
Audio Service	`getpid`, `ioctl`
Battery service	`close`, `gettid`, `open`
Location manager service	`getpid`, `ioctl`, `lseek`, `open`, `read`
Network policy management service	`getpid`, `ioctl`, `write`, `open`, `read`
Package manager	`close`, `getpid`, `gettid`, `ioctl`, `lstat64`, `open`, `write`, `socket`
Power manager service	`getpid`, `ioctl`, `open`, `read`, `recvfrom`
ServerThread	`close`, `connect`, `getpid`, `gettid`, `ioctl`, `lseek`, `lstat64`, `open`, `prctl`, `read`, `recvmsg`, `sendmsg`, `sendto`, `socket`, `write`
Wifi service	`getpid`, `gettid`, `send`, `recvfrom`
Window manager	`close`, `getpid`, `gettid`, `recvfrom`, `open`, `read`, `write`

then it reproduces the corresponding operation (`write`, `read` or `lseek`). Moreover, *KernelCallTester* handles `mkdir` and `rmdir` calls.
In each test, both with standard Android distributions and SEAndroid, we noticed that possible failures are only due to violations of the Linux DAC permissions (e.g. *KernelCallTester* is not in the owner group of a certain file). Those errors recur more often in SEAndroid test executions, since its policy explicitly labels the entire filesystem, preventing possible privacy leakages.

- **Process management system calls.** *KernelCallTester* is also able to successfully reproduce process management system calls like `gettid`, `getpid` or `exit_group`. Only the `kill` call cannot be reproduced because an unprivileged user is authorized to kill only process she owns. We did not recognized any difference between Android and SEAndroid.

Socket calls. Since sockets are mapped on files, *KernelCallTester* tries to `connect` to the socket before reproducing socket calls like `recvmsg` or `sendmsg`.
In standard Android, also in this case failures are due to insufficient Linux DAC permissions (e.g. sockets owned by *root* with permissions set to 660) or unaccepted parameters (e.g. `bind` fails because the targeted socket is already bound).
In SEAndroid, socket calls reproduced by *KernelCallTester* were blocked by the system and audited into the *dmesg* log system. This is due to the fact that the SEAndroid policy restricts the usage of sockets to trusted applications only.
In the following, an excerpt of *dmesg* log system is provided:

```
<5>[...] avc: denied {bind} for pid=3808 [...] scontext=u:r:
    untrusted_app:s0 tcontext=u:r:untrusted_app:s0 tclass=netlink_socket
```

In this example, the audit shows how a `bind` socket call to a netlink socket has been denied since the application is an `untrusted_app`.

Binder calls. Binder calls rely on the `ioctl` system call which allows sending simple commands and data to a file descriptor. Regarding IPC, the sender process performs a `ioctl` call to the Binder, specifying the addressee and including the message data. The Binder, which handles the passing mechanism, reads incoming messages and routes them to the appropriate destination. *KernelCallTester* is not able to reproduce exactly the same call since one of the arguments is the pointer to the data sent which is located within addresser's memory space, causing a *Bad Address* failure.
In SEAndroid besides such failure, several Binder transactions, and consequently `ioctl` calls, were blocked by the Intent MAC, as described in Section 4.2.

6.1. Analysis of results

Experiments show that *many* (85%) *kernel calls invoked by trusted services in the* AF *layer can be potentially reproduced by any unprivileged application.* Although unprivileged calls do not imply security threats per se, the lack of control over them is an undesired feature (e.g. DoS attacks).

The experimental results suggest that the native ASF does not recognize as malicious kernel call invocations attempted by an application in stead of a trusted service, revealing that native ASF is unable to discriminate according to the caller of a kernel call. In fact, as discussed above, failures in kernel call replication are due to Linux DAC permission errors or bad address issues, while the ASF never intervene when an application directly triggers the Linux kernel.

In addition, our empirical assessment shows that also SEAndroid, despite its security policy enforces restrictions on processes and objects, does not explicitly deal with kernel calls, only denying specific operations that violate the SELinux policy independently from the entity that tries to execute the invocation. For instance, the experiments show that socket calls are restricted to applications by the SELinux policy. This means that *the security enhancement granted by SEAndroid, albeit posing some limitations on the possibility to directly trigger the Linux Kernel, still lacks in systematically discriminating between the caller of kernel calls*. In this respect, SEAndroid keeps allowing applications to execute the main part of the supported kernel calls (i.e. the 75%). Moreover, both in SEAndroid and in standard distributions, no intervention is performed on repeated (and suspicious) invocation of the same kernel call; in particular, each attempts to replicate (also 10K times in a few seconds) the same kernel call is never recognized as suspicious. Thus, the possibility to properly forge and directly execute kernel calls may allow any application to exploit kernel level vulnerabilities.

To this aim, in the next section we discuss two kernel call-based interplay that a malicious application can execute to reduce the performance of the device and to violate the privacy of the user. We then discuss the security offered by both Android and SEAndroid against such interplay.

7. Kernel calls and malicious interplay

A central design point of the Android security architecture is that applications cannot adversely impact other applications and the operating system, or the user.[7] Such statement represents the final security goal of the Android operating system, according to its official documentation. Starting from this statement, we identified two security goals that should be obliviously granted:

- an application cannot exhaust system resources;
- an application cannot access data of another application.

Then, we analyzed all logs produced by the MKM during the testing phase in order to find potential vulnerabilities that, if exploited, allow to violate the previous security goals. The analysis has been carried out with both manual and automatic inspection of the whole set of logs.

Attacks on native Android. The analysis activity based on logs on Android v. 2.3.7 and v. 4.1.3 allowed to infer that:

- file system operations with proper parameters and sufficient permissions are unbounded;
- files located in `/data/data/com.android.browser/cache/webviewCache`, containing the cache (e.g. images, javascript codes, web pages accessed by the user) of installed browsers, can be read by any application.

Thus, we implemented two malicious applications, namely *WriteTest* (requiring no privileges upon installation) and *CacheHooker* (requiring only the INTERNET permission) that exploit such characteristics to perform attacks by means of kernel calls.

WriteTest repeatedly execute an `open`, a `write` and a `close` system call at different periods of time. Each interplay creates a dummy file of 4 MB in the internal memory of the phone.

CacheHooker cyclically invokes a `read` on the cache file, followed by a `write` on its data folder. Moreover it connects (using `socket` and `connect` calls) and sends (relying on `sendmsg` call) every copied file to a remote server. The connecting operation exploits the `android.permission.INTERNET` permission.

Attacks on SEAndroid. SEAndroid objects, e.g. files and sockets, are labeled by the `seapp_context` of the SELinux MAC policy. Consequently, *CacheHooker* is not able to read files located in `/data/data/com.android.browser/cache/webviewCache` as in the Android case. In fact the security policy blocks the opening attempt and logs a denied audit:

```
<5>[...] avc: denied {open} for pid=12433 [...] scontext=u:r:
    untrusted_app:s0 tcontext=u:r:untrusted_app:s0 tclass=cache_file
```

However, despite the restrictions imposed by the SEAndroid policy, the *WriteTest* malicious application can carry out the attack, leading to a saturation of the internal memory of both Nexus 4 and Nexus 5 devices.

We installed both applications to the testing smartphones of Section 6, so that they execute as services in background after a random period of time after the boot completion. Users reported that other applications had started to crash or to terminate at launch, and that Android provides back pop-ups on the exhaustion of the phone memory. Uninstalling other applications did not solve the problem. Only very few users were able to identify the *WriteTest* application and solve the problem by flashing the device. Furthermore, during the testing period *CacheHooker* has successfully and silently grabbed users browsing data in v 2.3.7 and v. 4.1.3, and, then, it has delivered all cached data to the remote server.

Discussion. The security weaknesses exploited by *CacheHooker* and *WriteTest* unveil some limitations in the ASF and SEAndroid.

[7] https://developer.android.com/guide/topics/security/security.html

CacheHooker is able to violate the sandbox of the native Android browser (i.e. WebKit), exploiting a development bug that allows stealing the browsing cache. Normally, an application that tries to access the browser cache of other applications through the AF layer is blocked, i.e. it would not be able to steal the cache. However, the possibility to bypass the AF layer through direct kernel call invocation frustrates the protection offered by both the native permission enforcement mechanism and the SEAndroid MMAC. Although the specific development bug has been recently fixed with a system update, this fact suggests that *direct* kernel call invocations could increase the exploitation of bugs and vulnerabilities.

WriteTest is able to exhaust the physical memory of the device through unbounded writing operations on the internal memory. Although the writing operation is not a malicious action per-se, its unbounded repetition leads to severely affect the usability of the device. This exploit is possible since Android does not pose any limitation to the operations performed by an application within its user space, both in primary and in secondary memory, as discussed in Example 1.

Therefore, neither the ASF and SEAndroid is able to prevent the repetition of operations. In fact, as shown in Example 2, the policy restricts actions allowed by subjects but it does not pose any limitation on its repetition. This fact opens up to the exploitation of development bugs to mount Denial-of-Services attacks as in the case of WriteTest and the vulnerability described in [1]. To this aim, the frequency of operations in Android should be monitored by the Android security mechanisms, especially in case of kernel calls.

8. Kernel call controller: a monitor for kernel call enforcement

We argue that malicious interplay related to kernel calls may be limited by providing the native ASF and SEAndroid with the possibility to identify both *the caller* of a kernel call and *the frequency* of kernel call invocations.

Android does not provide any native enforcement method to monitor kernel call invocations. On the other hand, the expressiveness on SELinux MAC policy language is coarse grained, e.g. it does not allow to deny a specific invocation based on a frequency analysis of which it occurs.

In order to overcome the limitations of both systems, in this section we introduce a Library module compatible with both Android and SEAndroid, called *Kernel Call Controller* (hereafter, KCC), able to limit the interaction between the Android stack and the Linux Kernel by enforcing a policy on kernel call invocations. In this respect, in this section we introduce the architecture of KCC and the language used to define KCC policies. By further analyzing malicious kernel call invocation, we propose a sample KCC policy and we test it both on Android and SEAndroid real devices. Finally, we empirically assess the performance of KCC as well as the efficacy of the approach using a sample policy.

8.1. Kernel call controller

The KCC is a library module that enriches the security capability of the ASF with the possibility to evaluate kernel call invocations. KCC is deployed as a library module to be included in the System Server process.

The KCC architecture is depicted in Fig. 4. In details, we extended the Bionic Libc by modifying the code that invokes kernel calls. Prior to the execution of a kernel call, Bionc Libc calls the enforcement point, i.e. the *KCC stub*, which invokes the *KCC library* placed in the System Server. The KCC library evaluates the compliance of the incoming kernel call against a set of enforcement rules, defined as a security policy (i.e. *KCC policy*), and reports back the KCC Stub which blocks or allows the call accordingly.

Furthermore, information collected by the KCC library are stored in the `/data/data` folder as a text file for offline analysis. The KCC policy can be modified by the *Activity Manager*.

The KCC can be adopted both on standard Android and SEAndroid distributions. Despite the latter already provides security hooks into the system call routines, as discussed in Section 4, the KCC does not affect the SEAndroid security model but, on the contrary, it can be transparently integrated in it.

8.2. KCC policy language

KCC enforces a set of rules over the kernel call invocations at runtime. Rules form a *KCC policy P* according to the following definition.

$$P \triangleq \{id_0 : \overline{R_0} \ldots, id_n : \overline{R_n}\}$$
$$R \triangleq \langle \alpha, q, d \rangle.$$

In words, a policy P is a set of labeled lists of rules. Also, we assume labels $id_0, \ldots, id_n$ appearing in a policy to be unique and we reserve the special label *fail*. Each rule R is a tuple $\langle \alpha, q, d \rangle$ where:

- $\alpha \in KC \times S \times O$ is an access action, i.e. a triple (a, s, o) where a is a kernel call, s is a subject and o is an object; both s and o can be omitted, meaning that the rule is applied to any subject and/or the object involving the kernel call a.
- $q \in \mathbb{Q}_0^+$ is a frequency value.
- $d \in \{id_0, \ldots, id_n, fail\}$ is a state transition label.

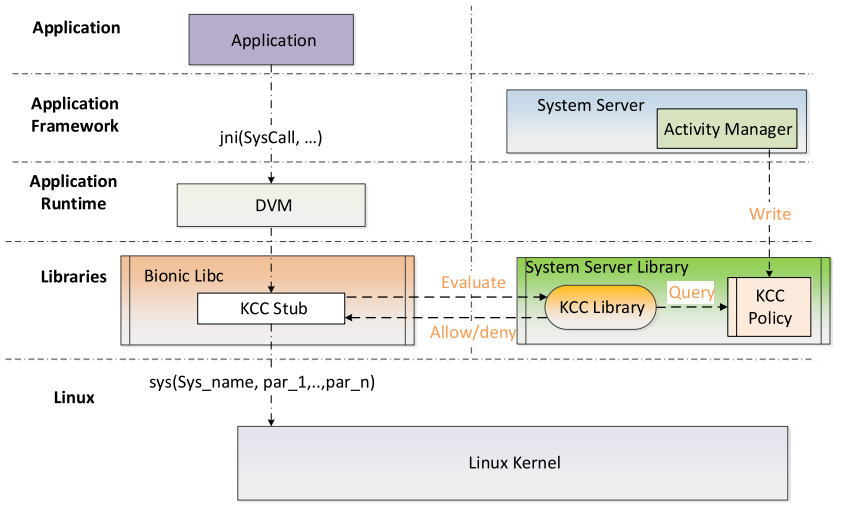

Fig. 4. The KCC architecture.

The meaning of a policy is that actions cause a change of state if they are observed more often than the given frequency. Notice that, if more than one rule apply, the policy considers each possible state transition (see below). Transitions may lead to the special, error state *fail* which denotes a policy violation. A trivial example of policy is $P = \{id_0 : \langle (write, s, o), 6, fail\rangle\}$, which states that the execution of more than six `write` per second by s on o is unauthorized by P.

The semantic of a policy P is given in terms of *Timed Security Automata* (TSA), a (slightly) modified version of timed automata [13].

A TSA $\mathcal{A}$ is a tuple $\langle \Lambda, \Sigma, \sigma_0, f, E\rangle$, where:

- Λ is a finite alphabet;
- Σ is a finite set of states, being σ_0 the initial one;
- $f \in \Sigma$ is the final state;
- $E \subseteq \Sigma \times \Lambda \times \mathbb{Q}_0^+ \times \Sigma$ is a finite set of timed transition (we write $\sigma \xrightarrow{\alpha,q} \sigma'$ for $\langle \sigma, \alpha, q, \sigma'\rangle \in E$).

The TSA is non-deterministic as an action can trigger more than one transition. As a consequence, the current configuration of a TSA is represented by a subset of its states. We write $FS(\sigma, \alpha, q) \in 2^{\Sigma}$ for the set of states reachable from σ with a transition α, q. In symbols

$$FS(\sigma, \alpha, q) = \{\sigma' \text{ s.t. } \exists\, \sigma \xrightarrow{\alpha,q'} \sigma' \wedge q' \leq q\}.$$

A run r of a TSA $\langle \Lambda, \Sigma, \sigma_0, f, E\rangle$ is a sequence of the form

$$S_0 \xrightarrow{\alpha_0,q_0} S_1 \xrightarrow{\alpha_1,q_1} S_1 \xrightarrow{\alpha_1,q_1} \ldots \xrightarrow{\alpha_{n-1},q_{n-1}} S_n$$

where $S_0 = \{\sigma_0\}$ and $\forall_{0\leq i<n}.S_{i+1} = \bigcup_{\sigma\in S_i} FS(\sigma, \alpha_i, q_i)$. Finally, r is a *violating trace* if $fail \in S_n$ or $S_n = \emptyset$.

We now describe the translation procedure from a policy P to a corresponding TSA $\mathcal{A}_P$. Let P be the policy defined as follows.

$$P = \begin{cases} id_1 : & R_1^1 & \ldots & R_{m_1}^1 \\ \quad\vdots & & & \\ id_n : & R_1^n & \ldots & R_{m_n}^n. \end{cases}$$

The corresponding TSA is $\mathcal{A}_P = \langle \Lambda, \Sigma, \sigma_0, f, E\rangle$ such that

- $\Lambda = \{\alpha | \exists i, j.R_j^i = \langle \alpha, q, d\rangle\}$;
- $\Sigma = \{\sigma_0, \ldots, \sigma_n, fail\}$;
- $f = fail$;
- $E = \{\sigma_i \xrightarrow{\alpha,q} \sigma_j | \exists i, k.R_k^i = \langle \alpha, q, id_j\rangle\} \cup \{\sigma_i \xrightarrow{\alpha,q} fail | \exists i, k.R_k^i = \langle \alpha, q, fail\rangle\}$.

To clarify this aspect we propose the following example.

Example 3. Consider the policy P defined below.

$$P = \begin{cases} id_0 : & \langle (\text{write}, 100, fd), 10, id_1\rangle \\ & \langle (\text{open}, 100, fd), 7, id_2\rangle \\ id_1 : & \langle (\text{open}, 100, fd), 10, fail\rangle \\ id_2 : & \langle (\text{close}, 100, fd), 3, id_1\rangle. \end{cases}$$

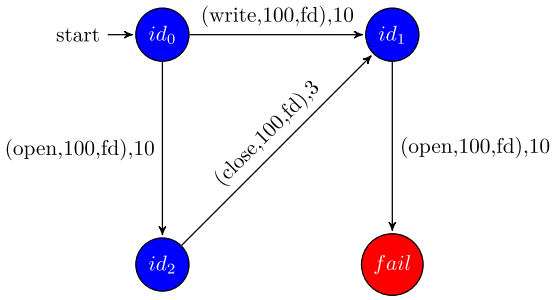

Fig. 5. TSA $\mathcal{A}_p$ for the sample policy P of Example 3.

In words, this policy limits the kernel call invocation on a specific file descriptor *fd* by the uid 100. *P* corresponds to the TSA $\mathcal{A}_P = \langle \Lambda, \Sigma, \sigma_0, f, E \rangle$ such that:

$$\mathbb{A}_p = \begin{cases} \Lambda = \{(\text{write}, 100, fd), (\text{open}, 100, fd), (\text{close}, 100, fd)\}; \\ \Sigma = \{id_0, id_1, id_2, \text{fail}\}; \\ \sigma_0 = id_0; \\ f = \text{fail}; \\ E = \text{is the set of transitions graphically depicted in Fig. 5}. \end{cases}$$

Notice that the KCC policy language is significantly more expressive than that for SELinux MAC policies. Indeed, apart from the missing notion of access frequency, SELinux relies on ACM (see Section 4.1), which are stateless, while KCC policies are history-based [14].

8.3. Defining a KCC policy for malicious invocations

We discuss here a KCC policy allowing to discriminate between normal and malicious behaviors in terms of kernel calls, according to (i) the identity of the caller and (ii) the number of repeated invocations. Unfortunately, current literature on system call analysis in Android (see Section 9) provides no useful indication. Therefore, we followed an empirical approach to infer suitable values for a policy definition, as detailed in the following.

Caller identity. We assume that Android services (e.g. services in the *AF* layer) are trusted (as the native ASF and SEAndroid policies do). This choice is suitable since Android services can be maliciously tampered only through device rooting.[8] According to literature [15], rooted distribution are untrusted by definition and any security measure is ineffective. Thus, we deal with unrooted distribution only and therefore we can assume each Android services as trusted. To this aim, our policy considers all direct invocations from Android services trusted as well as potentially malicious each kernel call invocation executed by an application directly (i.e. bypassing the AF layer).

In this respect, we analyzed Android v. 2.3.7, v. 4.1.3 and v. 4.4.4 to ascertain whether there is a way to clearly identify applications and AF services. Interestingly, the Linux Kernel in Android is configured to reserve a well-defined range of process identifiers (PIDs) to its system services. In detail, `android_filesystem_config.h`[9] associates each Android service with a specific PID, while reserving PIDs greater than 10,000 to applications only. Hence, our policy grants direct kernel call invocation only to PIDs lower than 10,000.

Kernel call replication. In order to infer a suitable upper bound value to kernel calls replication, we carried out another set of experiments, both on Android and SEAndroid, with the aim to profiling the system behavior during its standard execution (e.g. open the browser or send an SMS). Therefore, we extended the features of the MKM presented in Section 5 with the possibility to log the timing of each kernel call invocation. Then, we equipped the testing smartphones presented in Section 6 with the enhanced MKM and we log the type of call performed by each process in one day of activity. During this training period, we focused the testing on Android trusted services and pre-installed applications. This allows us to develop a basic profile for the expected behavior of the OS.

Fig. 6 shows excerpts of the number of kernel call invocations performed by Android processes during the training period. Interestingly, each process invokes less than 10 times per second the same type of kernel call, while, for instance, the *WriteTest* application performs hundred of `write` call per second. This behavior suggests that 10 replications per second can be a suitable upper bound to discriminate between legal and illegal kernel call replications.

KCC policy. The discussion regarding the identity of the caller of kernel call invocations and their replication over time, allows the definition of a sample KCC policy that (i) blocks all the kernel calls invocations performed by user processes (i.e. PID greater than 10,000) and (ii) restricts all other invocation to the upper bound of 10 times per second. For the sake

8 http://droidlessons.com/what-is-rooting-on-android-the-advantages-and-disadvantages/.

9 https://android.googlesource.com/platform/system/core/+/master/include/private/android_filesystem_config.h.

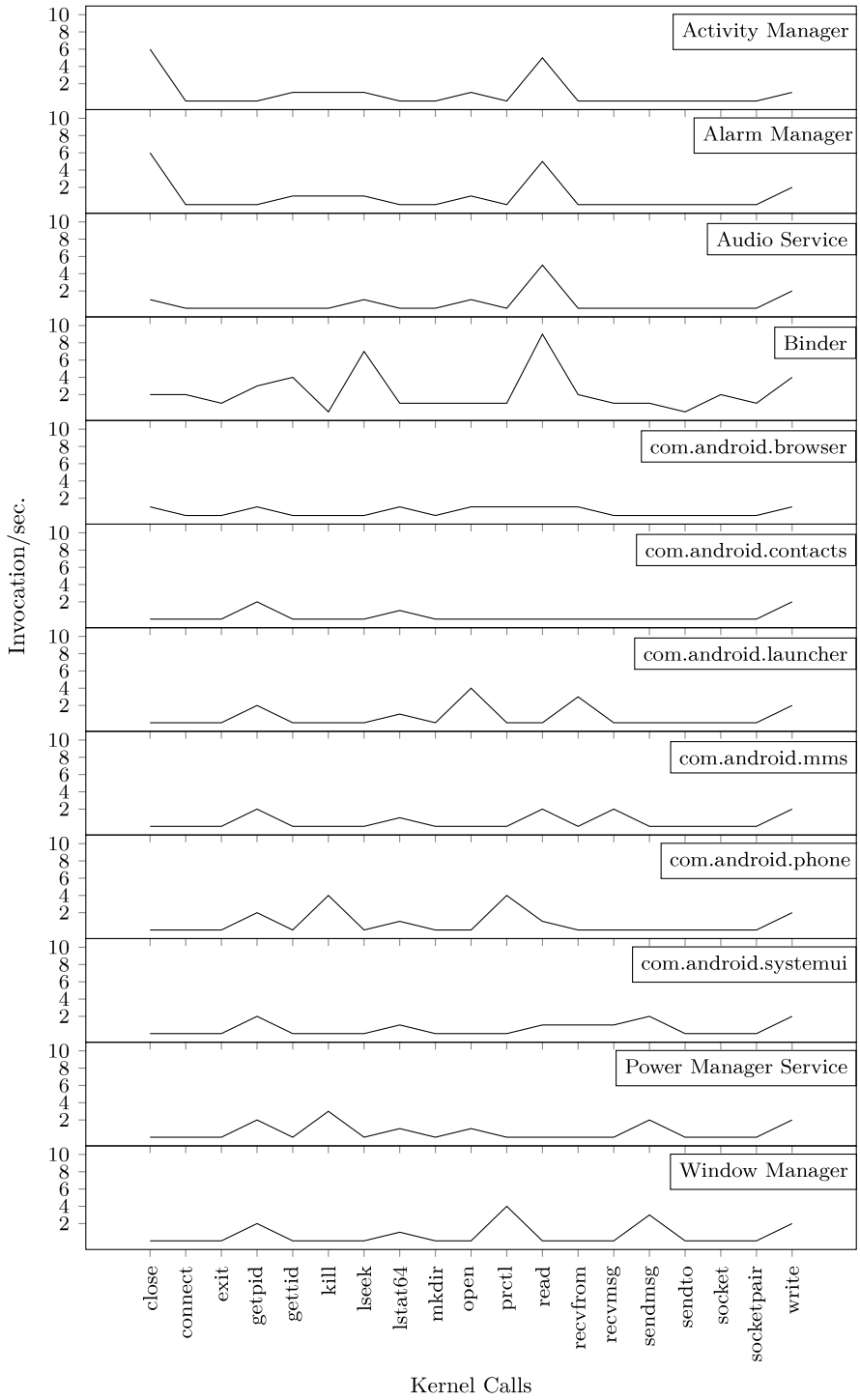

Fig. 6. Frequency of kernel call invocations in Android processes.

of brevity, a rule R can specify a generic kernel call a on an object o, meaning that such rule is replicated for each kernel call monitored by the KCC (listed in Section 5.2). Moreover, we defined the macros *SYS* and *USER* that represent, respectively, system PIDs (i.e. lower than 1000) and user PIDs.

$$P = \begin{cases} id_0 : & \langle (USER, a, o), 0, fail \rangle \\ id_1 : & \langle (SYS, a, o), 10, fail \rangle . \end{cases}$$

8.4. Effectiveness

We deployed KCC with the policy defined in Section 8.3, as well as *WriteTest* and *CacheHooker* in the customized Android versions (v. 2.3.7 and v. 4.1.3) and in the SEAndroid one (v. 4.4.4), and then we installed them into the thirteen smartphones previously used in our experiments (see Section 6). Then, we repeated the same test described in Section 6, providing users with the actual devices. We asked users to install up to 50 applications (if possible, according to the available resource on the device). For this test set we also deploy the KCC with the policy defined in Section 8.3.

KCC was able to block unexpected direct kernel call invocations from *WriteTest* and *CacheHooker* applications, thus fully preventing the memory exhaustion and the web cache privacy leak (i.e. no data has been sent to the remote server). During the testing phase, no user reported any visible performance issue or unexpected behavior.

Discussion. Experimental results suggest that applications may not need to perform direct kernel call invocations to work properly. However, further analysis should be carried out on a more comprehensive set of applications. In fact, we cannot exclude that some legal applications may require to interact directly with the Kernel without necessarily affecting the security of the Android distribution. In this respect, our policy may be too restrictive in the general case, since it blocks every direct invocation of kernel calls coming from a PID greater than 10,000. In this sense, we plan to extend the KCC with the possibility to identify applications in terms of their package name, in order to allow/deny the direct invocation of kernel calls also on a per-application basis. The limitation to 10 invocations per second avoids Denial-of-Service attacks to the Kernel and the hardware driver. Also in this case, the tests on the field confirmed the assessment discussed in Section 8.1 (i.e. trusted services in the AF layer execute less than 10 kernel call per second). However, the KCC policy language allows to easily customize previous rules in case they result to be inadequate, for instance, in other versions of the Android OS or to specific sets of applications.

8.5. Compatibility and performance evaluation

Using KCC to capture and analyze kernel calls at runtime may not comply with the Android standard policies defined by Google.[10] Furthermore, the runtime monitoring activity may impact on the performance of the devices. For these reasons, we carried out both a *compliance* and a *performance* evaluation of KCC.

Regarding the former point, we checked whether the deployment of KCC on a device is compatible with the Android standards defined by Google. To this aim, we executed the Compatibility Test Suite (CTS)[11] – designed to provide vendors with a benchmark to test the compatibility of their devices – on a top notch smartphone, i.e. Nexus 4 equipped with Android v. 4.4.4 (API 19) and SEAndroid, and an entry level-one, i.e. Samsung GT-S7500 with Android v. 2.3.7 (API 10). The CTS checks whether a given hardware/software configuration is compatible with Google standard policies. Both KCC-enabled devices passed the CTS testing. Clearly, knowing that KCC is aligned with the Android standards is not sufficient to guarantee the integrity of the modified OS. Indeed, a faulty implementation of the KCC might introduce low level, potentially disruptive, vulnerabilities. To cope with this scenario, some verification techniques can be applied, e.g., contract-driven development, model checking and testing. However, this is out of the scope of the present work and we postpone it to future developments.

On the same group of devices, we empirically assessed (through benchmarking) the degradation of performance due to the adoption of KCC. More specifically, we relied on three popular benchmark applications from Google Play, namely *AnTuTu*,[12] *Quadrant*[13] and *BenchmarkPi*,[14] to simulate the execution of general-purpose operations. AnTuTu and Quadrant are applications that evaluate the performance of a device by considering CPU, memory, I/O, 2D and 3D graphics. BenchmarkPi is mainly a CPU-intensive application that approximates the numerical value of π. Both AnTuTu and Quadrant benchmarks return a *score* (the higher the better) measuring the overall performance of the device. Instead, BenchmarkPi measures the *time* (the lower the better) in milliseconds required to complete the computation.

The histograms in Figs. 7 and 8 show the benchmark results for the standard Android OS (running on the Samsung GT-S7500) and for the SEAndroid OS (running on the Nexus 4), respectively. We tested the performance of each device according to three different configurations: (i) without KCC (Stock OS) (ii) with the KCC on the reduced set of kernel calls discussed in Section 5.2 (OS + KCC) and (iii) with KCC on the complete set of kernel calls (OS + Full KCC). As expected, performances decay in presence of (Full) KCC. In terms of percent decay, the scores for Android decrease by 11.3%–23.3% (AnTuTu) and 11.6%–22.3% (Quadrant), while BenchmarkPi time increases by 10.1%–41.5%. Similarly, for SEAndroid we observe that scores decrease by 8.8%–26.8% (AnTuTu) and 4.2%–23.8% (Quadrant), while CPU time increases by 15.0%–47.5%.

Summarizing, these results confirm that the KCC approach has a noticeable impact on the performances of the hosting device. Clearly, the effort for the computing platform grows with the number of observations, i.e., the number of monitored system calls. Reasonably, we expect real policies to refer to a security-relevant subset rather than to all of them. The investigation of possible optimizations making the KCC computation even lighter accounts as a future work.

10 http://static.googleusercontent.com/external_content/untrusted_dlcp/source.android.com/en/compatibility/4.2/android-4.2-cdd.pdf

11 http://source.android.com/compatibility/overview.html

12 http://www.antutu.com/en/index.shtml.

13 http://www.aurorasoftworks.com/products/quadrant.

14 http://androidbenchmark.com/.

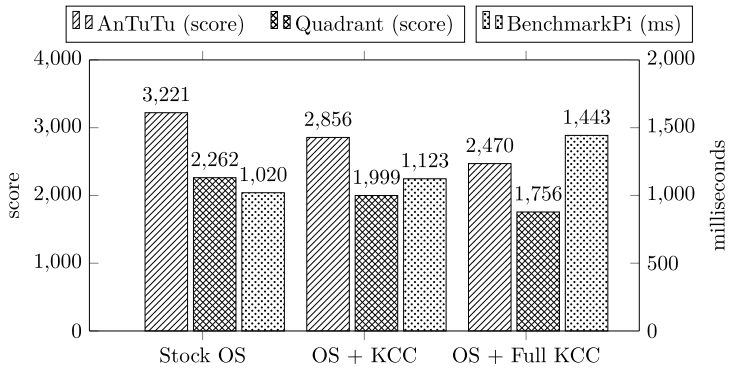

Fig. 7. KCC performance evaluation on Android OS (equipped on Samsung GT-S7500).

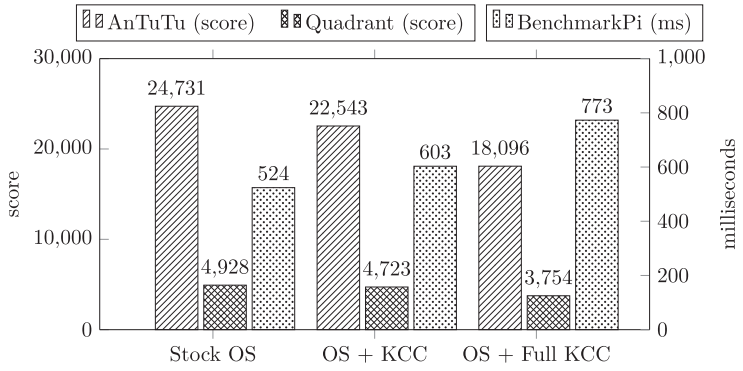

Fig. 8. KCC performance evaluation on SEAndroid OS (equipped on LG Nexus 4).

9. Related work

Literature on Android security has considerably spread in recent years, including general surveys, like [16–18], vulnerabilities, tools and formal methods to enhance both the Android architecture and the corresponding security model. Regarding vulnerabilities, recent work shows that the Android platform may suffer from DoS attacks [1], covert channels [19], web attacks [20] and privilege escalation attacks [2]. The same work underlines that such vulnerabilities affect each Android build.

Several authors underline the limitation of the current Android security model, thus proposing methodologies and tools to extend the native Android security solutions. For instance, in [21] authors propose an extension to the basic Android permission system and corresponding new policies, while in [22] authors suggest new privacy-related security policies for addressing security problems related to users' personal data. Other tools are devoted to malware detection (e.g. XManDroid [23] and Crowdroid [24]) and application certification (e.g. Scandroid [25] and Comdroid [3]).

However, none of such solutions takes into consideration interplay between the Android stack and the Linux kernel, focusing on interplay related to the Android stack only.

The idea of enhancing the security of mobile operating system, like Android, using SELinux has been extensively discussed in literature (see e.g. [26] and [27]), albeit the most comprehensive solution is the SEAndroid approach we discussed in this paper. Although SEAndroid is actually included in AOSP since version 4.0.3., few contributions inspect its security issues, and without extensively testing the SEAndroid implementation. For instance, [28] provides an architectural overview of SEAndroid, while [29] proposes an extension of SEAndroid type enforcement called Flaskdroid. Finally, [30] discusses and exploits several weaknesses of current SEAndroid implementation.

The idea of monitoring system calls in mobile devices has gained momentum in very recent years. Most of the emerging solutions aimed at monitoring and analyzing system calls are devoted to malware detection. More specifically, system call tracing is a new approach to detect and identify malware in mobile devices. The idea is to profile malware sample system call invocations and compare them to applications under test in order to find similarities, in a signature-based fashion.

For instance, [31] proposes an Android Application Sandbox (AASandbox) able to perform both static and dynamic analysis in a fully isolated environment. AASandbox relies on a loadable kernel module which monitors system calls. In [32] authors analyze system call event patterns of top-selling Android games, comparing them with those extracted from 1260 malware samples distributed by the Android MalGenome Project. Albeit the methodology is sound, experimental results do not clarify how many games can actually embed known malware.

Authors in [33] propose DroidTrace, a dynamic analysis system used to explore the behavior of dynamically loaded code in Android applications. DroidTrace provides analysis results and reports allowing analysts to ascertain whether a given application has a suspicious behavior and, in case, intervene. Another tool for behavioral dynamic analysis of Android malware is CopperDroid [34]. CopperDroid automatically describes low-level OS-specific and high-level Android-specific behaviors of Android malware by observing and analyzing system call invocations. However, the effectiveness of this approach is severely limited by the number of malware system call patterns that are currently available. Moreover, this solution hardly copes with zero-day malware. Other tools, like ANANAS [35], DroidRanger [36] or DroidScope [37] include in their framework a module for auditing system call invocations. Although their implementation is similar to our proposal, these approaches are rather different. In fact, none of the those solutions has been adopted to perform a security assessment on Android kernel calls. On the contrary, also in this case their analysis is limited to the identification of well-known malicious application patterns.

Beyond signature-based malware detection, other approaches allow to monitor system call invocation and enforce security policies on them at runtime. For instance, authors in [38] propose FireDroid, a policy-based framework that instruments native system calls in order to enforce security policies on them. Albeit the solution is rather invasive in the Linux kernel, the main advantage of FireDroid is that it is completely transparent to the applications as well as to the Android stack. FireDroid also provides a specification language to define security policies: however, as the same authors state, "specifying security policies at the level of system calls is quite a tedious task" [38], and, hence, defining working and reliable security policy in FireDroid is still complex and cumbersome. As a matter of fact, the paper proposes two rather simple policies and there is no assessment on the complexity of policies that FireDroid can actually support.

Aurasium [39] is an application-hardening service that enforces security and privacy policies to arbitrary applications. Aurasium builds up a sandbox on each executing application and monitors its behavior in terms of system call invocation. The sandbox allows Aurasium to apply security policies at runtime, thereby allowing/denying single system call invocations. Differently from FireDroid, Aurasium does not support any specification language for security policies; on the contrary, it provides a set of pre-defined policies. Finally, it is worth noticing that the policy enforcement performed by both FireDroid and Aurasium may lead to application crashes.

To the best of our knowledge, our work is the first attempt to empirically investigate correlations and security issues related to the interplay among the Android layers and the Linux kernel. Furthermore, our solution allows to limit the operations specifically targeting the Linux Kernel, independently from nature of the invoking applications (i.e. trusted application/malware). More specifically, the current KCC implementation can be customized to limit the access to the Kernel according to the kind of calls or the identity of the caller.

10. Conclusions

In this paper the empirical methodology originally proposed in [5] has been extended to assess the efficacy of SEAndroid. An experimental environment for both Android and SEAndroid has been setup by building up a kernel module (i.e. the *Monitoring Kernel Module*) and an application (i.e. *KernelCallTester*) to capture and replicate all kernel calls invoked by trusted services in AF layer. Empirical analysis and comparison on the detection capabilities of both Android and SEAndroid environments have shown that also SEAndroid exercises very little control on the interactions between applications and the Linux kernel, opening up to potentially malicious and harmful interactions. To prove this claim, two malicious applications (i.e. *WriteTest* and *CacheHooker*), directly interacting with the Linux kernel, have been implemented. Then, their behavior has been tested both on Android and SEAndroid.

Finally, a library for run-time policy enforcement (i.e. *Kernel Call Controller* (KCC)) has been presented along with a language for policy definition. KCC enforces policies for detecting and limiting the direct interaction between applications and the kernel by the means of kernel calls. Furthermore, KCC has been validated against a sample policy. Finally, some efficacy and performance evaluations on KCC have been carried out.

The results discussed in this paper indicate that, despite the introduction of SEAndroid, Android is potentially still weak against direct attacks to the Linux kernel. In other words, this amounts to say that vulnerabilities in the Linux kernel can be easily exploited by malicious applications, since the limitation adopted by actual Android security mechanisms are rather inadequate. Our experiments suggest that the introduction of the KCC module in the Linux kernel allows to mitigate this problem without affecting the Android and SEAndroid design as well as the everyday user experience.

References

[1] A. Armando, A. Merlo, M. Migliardi, L. Verderame, Would you mind forking this process? A denial of service attack on Android (and some countermeasures), in: Proc. of the 27th IFIP International Information Security and Privacy Conference, SEC 2012, in: IFIP Advances in Information and Communication Technology, vol. 376, Springer, 2012, pp. 13–24.
[2] L. Davi, A. Dmitrienko, A.-R. Sadeghi, M. Winandy, Privilege escalation attacks on android, in: M. Burmester, G. Tsudik, S. Magliveras, I. Ilic (Eds.), Information Security, in: LNCS, vol. 6531, Springer-Verlag, 2011, pp. 346–360.
[3] E. Chin, A.P. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in android, in: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys'11, ACM, New York, NY, USA, 2011, pp. 239–252.
[4] A. Armando, A. Merlo, M. Migliardi, L. Verderame, Breaking and fixing the android launching flow, Comput. Secur. 39 (2013) 104–115.
[5] A. Armando, A. Merlo, L. Verderame, An empirical evaluation of the android security framework, in: Security and Privacy Protection in Information Processing Systems, in: IFIP Advances in Information and Communication Technology, vol. 405, Springer, Berlin, Heidelberg, 2013, pp. 176–189.
[6] National Security Agency, Security Enhancements (SE) for Android, http://seandroid.bitbucket.org/.

[7] S. Smalley, R. Craig, Security enhanced (SE) android: Bringing flexible MAC to android, in: NDSS, 2013.
[8] P. Brady, Anatomy and physiology of an android, Google I/O. https://sites.google.com/site/io/anatomy–physiology-of-an-android.
[9] S. Smalley, C. Vance, W. Salamon, Implementing selinux as a linux security module, Tech. Rep., Nai labs report, NAI Labs, 2006, https://www.nsa.gov/research/_files/selinux/papers/module.pdf.
[10] C. Wright, C. Cowan, S. Smalley, J. Morris, G. Kroah-Hartman, Linux security module framework, in: 11Th Ottawa Linux Symposium, 2002.
[11] S. Smalley, Middleware MAC for android, http://kernsec.org/files/LSS2012-MiddlewareMAC.pdf (August 2012).
[12] S. Krahmer, C-skills: Zimperlich sources., http://c-skills.blogspot.com/2011/02/zimperlich-sources.html.
[13] R. Alur, D.L. Dill, A theory of timed automata, Theoret. Comput. Sci. 126 (1994) 183–235.
[14] M. Abadi, C. Fournet, Access Control based on Execution History, in: In Proceedings of the 10th Annual Network and Distributed System Security Symposium, 2003, pp. 107–121.
[15] Z. Zhang, Y. Wang, J. Jing, Q. Wang, L. Lei, Once root always a threat: Analyzing the security threats of android permission system, in: Information Security and Privacy 8544 LNCS, 2014, pp. 354–369.
[16] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M.S. Gaur, M. Conti, M. Rajarajan, Android security: A survey of issues, malware penetration, and defenses, Commun. Surv. Tutor. 17 (2) (2015) 998–1022. http://dx.doi.org/10.1109/COMST.2014.2386139.
[17] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, C. Glezer, Google android: A comprehensive security assessment, IEEE Secur. Priv. 8 (2) (2010) 35–44.
[18] W. Enck, M. Ongtang, P. McDaniel, Understanding android security, Secur. Priv. 7 (1) (2009) 50–57.
[19] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, X. Wang, Soundcomber: A stealthy and context-aware sound trojan for smartphones, in: Proceedings of the 18th Annual Network & Distributed System Security Symposium, NDSS, 2011, pp. 17–33. http://www.isoc.org/isoc/conferences/ndss/11/pdf/1_1.pdf.
[20] T. Luo, H. Hao, W. Du, Y. Wang, H. Yin, Attacks on webview in the android system, in: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC'11, ACM, New York, NY, USA, 2011, pp. 343–352.
[21] M. Nauman, S. Khan, 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, ASIACCS'10, ACM, New York, NY, USA, 2010, pp. 328–332.
[22] Y. Zhou, X. Zhang, X. Jiang, V.W. Freeh, Taming information-stealing smartphone applications (on android), in: Proceedings of the 4th International Conference on Trust and Trustworthy Computing, TRUST'11, 2011, pp. 93–107.
[23] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, Xmandroid: A New Android Evolution to Mitigate Privilege Escalation Attacks, Tech. Rep. TR-2011-04, Technische Univ. Darmstadt, 2011.
[24] I. Burguera, U. Zurutuza, S. Nadjm-Therani, Crowdroid: behavior-based malware detection system for android, in: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, SPSM'11, 2011, pp. 15–26.
[25] A.P. Fuchs, A. Chaudhuri, J.S. Foster, Scandroid: Automated security certification of android applications, Manuscript, Univ. of Maryland, http://www.cs.umd.edu/~avik/projects/scandroidascaa, http://www.cs.umd.edu/~avik/projects/scandroidascaa/paper.pdf.
[26] A. Shabtai, Y. Fledel, Y. Elovici, Securing android-powered mobile devices using selinux, IEEE Secur. Priv. 8 (3) (2010) 36–44.
[27] D. Muthukumaran, J. Schiffman, M. Hassan, A. Sawani, V. Rao, T. Jaeger, Protecting the integrity of trusted applications in mobile phone systems, Secur. Commun. Netw. 4 (6) (2011) 633–650.
[28] C. Zheng, Overview of security enhanced android's security architecture, in: 2nd International Conference on Teaching and Computational Science, Atlantis Press, 2014, pp. 48–50.
[29] S. Bugiel, S. Heuser, A.-R. Sadeghi, Towards a framework for android security modules: Extending seandroid type enforcement to android middleware, Tech. Rep., Tech. Rep. TUD-CS-2012-0231, Center for Advanced Security Research Darmstadt, 2012.
[30] P.O. Fora, Defeating Security Enhacements (SE) for Android, https://www.defcon.org/images/defcon-21/dc-21-presentations/Fora/DEFCON-21-Fora-Defeating-SEAndroid.pdf (August 2013).
[31] T. Blasing, L. Batyuk, A.-D. Schmidt, S. Camtepe, S. Albayrak, An android application sandbox system for suspicious software detection, in: Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on, 2010, pp. 55–62.
[32] J.H. You, M. Daeyeol, L. Hyung-Woo, L. Jae Deok, K. Jeong Nyeo, Android mobile application system call event pattern analysis for determination of malicious attack, Int. J. Secur. Appl. (2014) 231–246.
[33] M. Zheng, M. Sun, J. Lui, Droidtrace: A ptrace based android dynamic analysis system with forward execution capability, in: Wireless Communications and Mobile Computing Conference, IWCMC, 2014 International, 2014, pp. 128–133.
[34] A. Reina, A. Fattori, L. Cavallaro, A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors, in: ACM European Workshop on Systems Security, EuroSec, ACM, 2013.
[35] T. Eder, M. Rodler, D. Vymazal, M. Zeilinger, Ananas—a framework for analyzing android applications, Proceedings—2013 International Conference on Availability, Reliability and Security, ARES, 2013 (2013) 711–719.
[36] Y. Zhou, Z. Wang, W. Zhou, X. Jiang, Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets, in: NDSS, 2012.
[37] L.K. Yan, H. Yin, Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis, in: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12), USENIX, Bellevue, WA, 2012, pp. 569–584.
[38] G. Russello, A.B. Jimenez, H. Naderi, W. van der Mark, Firedroid: hardening security in almost-stock android, in: Proceedings of the 29th Annual Computer Security Applications Conference, ACSAC'13, ACM, 2013, pp. 319–328.
[39] R. Xu, H. Saïdi, R. Anderson, Aurasium: Practical policy enforcement for android applications, in: Proceedings of the 21st USENIX Conference on Security Symposium, Security'12, USENIX Association, 2012, pp. 27–41.