AppPolicyModules: Mandatory Access Control for Third-Party Apps

Enrico Bacis
enrico.bacis@unibg.it

Simone Mutti
simone.mutti@unibg.it

Stefano Paraboschi
parabosc@unibg.it

Università degli Studi di Bergamo, Italy
Department of Management, Information and Production Engineering

ABSTRACT

Android has recently introduced the support for Mandatory Access Control, which extends previous security services relying on the Android Permission Framework and on the kernel-level Discretionary Access Control. This extension has been obtained with the use of SELinux and its adaptation to Android (SEAndroid). Currently, the use of the MAC model is limited to the protection of system resources. All the apps that are installed by users fall in a single undifferentiated domain, *untrusted_app*. We propose an extension of the architecture that permits to associate with each app a dedicated MAC policy, contained in a dedicated *appPolicyModule*, in order to protect app resources even from malware with root privileges.

A crucial difference with respect to the support for policy modules already available in some SELinux implementations is the need to constrain the policies in order to guarantee that an app policy is not able to manipulate the system policy. We present the security requirements that have to be satisfied by the support for modules and show that our solution satisfies these requirements. The support for appPolicyModules can also be the basis for the automatic generation of policies, with a stricter enforcement of Android permissions. A prototype has been implemented and experimental results show a minimal performance overhead for app installation and runtime.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access control

Keywords

Android, App Security, Policy Modularity, Administrative Policies, Mandatory Access Control, SELinux

1. INTRODUCTION

Mobile operating systems play a central role in the evolution of Information and Communication Technologies. One of the clearest trends of the past few years has been the adoption by users of mobile portable devices, replacing personal computers as the reference platform for the delivery of many ICT resources and services. The rapid success and wide deployment of mobile operating systems has also introduced a number of challenging security requirements, making explicit the need for an improvement of security technology.

The mobile scenario is indeed characterized by two mutually reinforcing aspects. On one hand, mobile devices are high-value targets, since they offer a direct financial incentive in the use of the credit that can be associated with the device or in the abuse of the available payment services (e.g., Google Wallet, telephone credit and mobile banking) [13,18]. In addition, mobile devices permit the recovery of large collections of personal information and are the target of choice if an adversary wants to monitor the location and behavior of an individual. On the other hand, the system presents a high exposure, with users of modern mobile devices continuously adding new *apps* to their devices, to support a large variety of functions (we follow the common convention and use the term *app* to denote applications for a mobile operating system).

The risks are then greater and different from those of classical operating systems [2]. The frequent installation of external code creates an important threat. The design of security solutions for mobile operating systems has to consider a careful balance between, on one side, the need for users to easily extend with unpredictable apps the set of functions of the system and, on the other side, the need for the system to be protected from potentially malicious apps.

It is to note that the greatest threats derive from apps that are offered through delivery channels that are alternative to the "official" *app markets* (e.g., [19]), whose number of app installations is increasing rapidly, pointing out the need of wider security layers. Apps in official markets, instead, are verified by the market owner and the ones detected as misbehaving are promptly removed from the market. The correct management of the app market is crucial, nevertheless it is not able by itself to fully mitigate the security concerns. The mobile operating systems have to provide a line of defense internal to the device against apps that, due to malicious intent or the presence of flaws in system components or other apps, may let an adversary abuse the system.

1.1 Rationale of the approach

The approach that we propose follows the principles of the Android security model, which aims at isolating from each other the apps that are executed by the system. Each app is confined within an assigned domain and interaction between

ASIA CCS'15, April 14–17, 2015, Singapore..
Copyright © 2015 ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714626.

the elements of the system is managed by a privileged component, which enforces the restrictions specified by a policy. The approach presented in the paper aims at strengthening this barrier, introducing an additional mechanism to guarantee that apps are isolated and cannot manipulate the behavior of other apps. The additional mechanism is obtained with an adaptation of the services of a Mandatory Access Control (MAC) model, which enriches the Discretionary Access Control (DAC) services native to the Linux kernel.

MAC models are commonly perceived as offering a significant contribution to the security of systems. However, one drawback of MAC models is represented by *policy management* which is a especially critical in complex systems such as Android, where each OEM tries to customize the MAC policy for its own devices. Samsung KNOX is the most well-known example. Policy customization provides benefits in terms of security, but it inevitably leads to *policy fragmentation*. Our work tries to do a step ahead in the policy standardization defining a set of *entry points* which can be used by both OEMs and developers in order to extend, under specific constraints, the MAC policy to fulfill their own security requirements and subsequently try to mitigate the policy fragmentation problem.

Apps can only become known to the system when the owner asks for their installation. The MAC policy has then to be dynamic, with the ability to react to the installation and deletion of apps, which requires modularity and the capability to incrementally update the security policy, with a policy module associated with an app. We use the term appPolicyModule to characterize it (when space is limited, like in table headers, we may use the acronym APM). The support for appPolicyModules allows app developers to benefit from the presence of a MAC model, letting them define security policies that increase the protection the app can get against attacks coming from other apps, which may try to manipulate the app and exploit its vulnerabilities.

1.2 Outline

Section 2 provides an overview of the Android security architecture, describing the role of the MAC model introduced by SEAndroid. Section 3 describes the threat to third-party apps that the policy modules want to mitigate. Section 4 presents a model of SELinux policies, used in Section 5 to formalize the requirements that policy modules have to satisfy. Section 6 introduces the syntax used by appPolicyModules. Section 7 illustrates how the use of appPolicyModules can improve the support of Android permissions. In Section 8 we discuss the performance results. Section 9 provides a comparison with previous work in the area. Finally, Section 10 draws a few concluding remarks.

2. ANDROID SECURITY ARCHITECTURE

The Android security model shows a direct correspondence with the overall Android architecture, which is organized in three layers (from bottom to top): (a) an underlying Linux kernel, (b) a middleware framework, and (c) an upper application layer. The Linux kernel in the lowest layer provides low-level services and device drivers to other layers and it differs from a traditional Linux kernel, because it aims at running in an embedded environment and does not have all the features of a traditional Linux distribution. The second layer, the middleware framework, is composed of native Android libraries, runtime modules (e.g., the Dalvik Virtual Machine and the alternative Android Runtime ART) and an application support framework. The third layer is composed by apps. Apps are divided into two categories: (i) pre-installed apps (e.g., Web browser, phone dialer) and (ii) *third-party* apps installed by the user. In the paper we focus on the consideration of third-party apps, since the pre-installed ones are already covered by the system policy.

Android provides distinct security mechanisms at the distinct layers. The Linux security model based on user identifiers (*uid*) and group identifiers (*gid*) operates at the lowest layer, with each app receiving a dedicated *uid* and *gid*. The granularity of this access control model is at the level of files and processes, reusing all the features of the classical DAC model of Unix/Linux, with a compact *acl* that describes for each resource the operations permitted respectively to the owner, members of the resource group, and every user of the system.

At the application layer, Android uses fine-grained permissions to allow apps or components to interact with other apps/components or critical resources. The *Android Permission Framework* contains a rich and structured collection of privileges, in the 4.4.4 version more than 200, focused on the management of the large variety of resources that are offered by the operating system to apps. The access control model assumes that apps specify in their manifest the set of privileges that will be required for their execution. At installation time, users of Android devices have to explicitly accept the request for privileges by the app; in case the user does not accept the app request, the app is not installed.

2.1 SEAndroid

Recently, the SEAndroid initiative [15] has led to a significant extension of the security services, with the integration of Security Enhanced Linux (SELinux) [12] into the Android operating system. The goal of SEAndroid is to build a *mandatory access control* (MAC) model in Android using SELinux to enforce kernel-level MAC, introducing a set of middleware MAC extensions to the Android Permission Framework. SELinux originally started as the Flux Advanced Security Kernel (FLASK) [10] development by the Utah University Flux team and the US Department of Defense. The development was enhanced by the NSA and released as open source software. SELinux policies are expressed at the level of *security context* (also known as *security label* or just *label*). SELinux requires a security context to be associated with every process (or subject) and object, which is used to decide whether access is allowed or not as defined by the policy. Every request a process generates to access a resource will be accepted only if it is authorized by both the classical DAC access control service and by the SELinux policy. The advantage of SELinux compared to the DAC model are its flexibility (the design of Linux assumes a *root* user that has full access to DAC-protected resources) and the fact that process and resource labels can be assigned and updated in a way that is specified at system level by the SELinux policy (in the DAC model owners are able to fully control the resources).

The middleware MAC extension chosen to bridge the gap between SELinux and the Android permission framework is called *install-time MAC* [15] (several middleware MACs have been developed, but only the install-time MAC has been integrated into the AOSP). This mechanism allows to check an app against a MAC policy (i.e., *mac_permissions.xml*).

The integration of this middleware MAC ensures that the policy checks are unbypassable and always applied when apps are installed and when they are loaded during system startup.

The current design of SEAndroid aims at protecting core system resources from possible flaws in the implementation of security in the Android Permission Framework or at the DAC level. The exploitation of vulnerabilities becomes harder due to the constraints on privilege escalation that are introduced by SELinux. Unfortunately, the current use of SELinux in Android aims at protecting the system components and trusted apps from abuses by third-party apps. All the third-party apps fall within a single *untrusted_app* domain and an app interested in getting protection from other apps or from internal vulnerabilities can only rely on Android permissions and the Linux DAC support. This is a significant limitation, since apps can get a concrete benefit from the specification of their own policy.

3. THREAT MODEL

In Android each app receives a dedicated *uid* and *gid* at install-time. These identifiers are used to set the user and the group owner of the resources installed by the app in the default data directory, which is */data/data/"package_name"*. By default, the apps databases, settings, and all other data go there. Since user data for an application also resides in */data/data/"package_name"*, it is important that only that application has access to that particular folder. This confinement of the data folders permits to enforce a strict isolation from other applications. In Android this isolation is only enforced at DAC level, but this is not enough to protect the app and its own resources by other apps with *root* privileges. Android, by default, comes with a restricted set of permissions for its user and the installed applications (i.e., no root privileges). Despite this, apps can gain root privileges in two ways and use it to provide desirable additional features for users, but a malicious app may also abuse it to bypass Android's security measures.

On one hand many benign apps require root privileges to accomplish their job. For example, *Titanium Backup* [17] is one of Google Play's best-selling apps and it needs root privileges to backup system and user applications along with their data. In this scenario, the user typically flashes a recovery console on the device which has the permission to write on the system partition and from there she installs an app such as *SuperSU* or *Superuser* in order to gain and manage root privileges. After that, the user can give root privileges to other applications. According to Google, users install non-malicious rooting apps by a ratio of 671 per million in 2014 (increased by 38% compared to the 491 per million in 2013 [11]). Moreover, there are successful community-ROMs, such as CyanogenMod with over 10 million installations, that provide root access to the user by default. Here, the user is aware of the fact that some apps act as root in the system and have access to everything, however she does not know how these privileges are used and she has to trust the app.

On the other hand, a malware could exploit a bug in a system component and gain root privileges to freely access the whole system in order to steal personal information or perform fraudulent actions. In this scenario the user is unaware of the fact that an app acts as root. Over the years Android has been attacked by threatening malware apps such as *DroidDreamLight*, which affected 30,000-120,000 users in May 2011 [3]. Recently the app *towelroot* has been released which, exploiting the *CVE-2014-3153* bug of Linux kernel, permits to "root the device" without the need to flash a recovery console, and gives root privileges potentially to all apps. This bug affects all Android versions up to 4.4.4 and thus represents a significant threat in the current DAC-only protection of private app resources.

Our proposal provides a solution to both scenarios through the use of appPolicyModules defined by the app and attached to the SELinux system policy.

3.1 Example

Hereinafter, we define a running example that we use as a proof of concept of our solution. *Dolphin Browser* [7] is considered one of the most successful mobile browsers for Android[1] with over 100 million downloads. It uses the *Webkit* engine and provides several features such as *gesture browsing* and *browsing boost*. We use it to show how the threat model defined in the previous section affects the current DAC-only security isolation of a real app and its private data. In Section 5 we use this example to identify the requirements and then we will illustrate how the use of a dedicated appPolicyModule can provide better security for its private data. The Android permissions requested by the app are:

```
android.permission-group.NETWORK
android.permission-group.ACCOUNTS
android.permission-group.LOCATION
android.permission-group.MICROPHONE
android.permission-group.CAMERA
```

Many browsers include a password manager component that stores confidential information such as usernames and passwords. The common strategy used by almost all the mobile browsers we have analyzed is to keep the credentials in a SQLite database. Following Google's best practices for developing secure apps, the password database is saved in the app data folder, which should be accessible only to the app itself. Another best practice (not used by Dolphin browser) to provide additional protection for sensitive data, is to encrypt local files using a key that is not directly accessible to the application. For example, a key can be placed in a *KeyStore* and protected with a user password that is not stored on the device. While this does not protect data from a root compromise that can monitor the user inputting the password, it can provide protection for a lost device without file system encryption.

Some of the browsers we have analyzed (e.g., *Google Chrome*) store the passwords in plaintext in the database, while others use some form of encryption (e.g., *Dolphin Browser*, *Firefox*). The decision to keep the passwords in plaintext can appear as a weakness, but even when the information is stored in an encrypted form, if the data needs to be recovered automatically by the app without the need of addi-

[1]At the time of writing *Chrome* is the most used mobile browser for Android devices; however, due to the fact that *Chrome* is included in the *Gapps*, it does not belong to the *untrusted_app* domain but to the *isolated_app* domain, thus it is not considered as a third-party app. The same discussion and threat model presented for Dolphin Browser is also valid for *Google Chrome for Android*. Moreover, all the passwords that the user saved in the Desktop version of Google Chrome using the same login details are available in the Android database.

tional information not stored on the device (e.g., a master password known only by the user), a malware could use the same resources used by the legitimate application to retrieve the information.

There are a number of ways one can obtain the Java code back from the APK in order to study the app behavior, to replicate it and to extract the encrypted information. To encrypt the passwords, *Dolphin Browser* used to adopt a static key, which was obtainable by simply looking at the decompiled bytecode. Newer versions of the browser derive the key from the *android_id* of the device, generated during the first boot, whose use is encouraged by the Android Developers community to generate device-specific passwords.

We were able to obtain the decrypted passwords from the *password.db* decompiling the app and studying its behavior. In the same way, a malware that managed to obtain root privileges can access the database and decrypt all the user credentials.

4. SELINUX POLICY MODEL

SELinux uses a *closed* policy model, denying every access request that is not explicitly permitted. The SELinux policy is defined using rules, which produce a set of authorizations. The SELinux model is quite rich and offers a number of features that increase its expressive power and flexibility. For instance, SELinux is able to manage a Multi-Level Security model, with the representation of sensitivity labels and categories. These features are used in some systems that rely on SELinux (e.g., Samsung Knox), but they are not currently used in AOSP, which is our reference platform. We then propose a simpler model that allows us to better characterize our approach.

The model uses names with an "av" prefix, like avType instead of "type", to provide a more precise definition. In the remainder of the paper, we will sometimes used the simpler terms (i.e., type instead of avType) when we see no ambiguity in their use. The "av" prefix stands for "access vector" and is used in SELinux to characterize the rules defining the policy, called AV Rules. The basic elements of this model are:

avType: represents an identifier that can be used to describe both the subject and the target of an authorization; an avType denotes a security domain or the profile of a process or resource in the system; the avType is used to build labels for processes and resources.

avClass: represents the kind of resource (e.g., file, process) that will be the target of an authorization; an implementation of SELinux in a system will have to provide in its setup a set of avClasses consistent with the variety of resources that the system is able to manage.

avPermission: represents the possible actions a source can apply on a target of a specific avClass, specified in the setup of SELinux; every avClass *cl* has its own set of avPermissions, represented by *cl.permissions* (e.g., $file.permissions = \{read, write, execute, \ldots\}$).

In order to give a formal representation of the SELinux policy, we introduce the concept of avAuthorization.

DEFINITION 1. *Given a set T of avTypes, a set C of avClasses, and a set P of avPermissions, an* ***avAuthorization*** *a is a quadruple* ⟨ source, target, class, action ⟩, *where:*

source $\in T$ *represents the process (the security principal of the authorization);*

target $\in T$ *is associated with the object that is accessed by the* source*;*

class $\in C$ *denotes the type of resource that is accessed in the operation;*

action $\in \{P \cap \text{class.permissions}\}$ *is the specific avPermission, which has to be compatible with the avClass.*

Each avAuthorization describes a specific request that is permitted in the system.

EXAMPLE 1. *Consider an app whose process is associated with the avType myapp that wants to read a file both in the internal and external sdcard. The required avAuthorizations are as follows:* ⟨ myapp, sdcard_internal, file, read ⟩, ⟨ myapp, sdcard_external, file, read ⟩.

DEFINITION 2. *An* **avAuthzPolicy** *is a set of avAuthorizations.*

The avAuthzPolicy is derived from the specification of a collection of *avRules.* avRules can be positive or negative, support the use of patterns for the specification of sources and targets, and may use avAttributes.

DEFINITION 3. *An* **avAttribute** *is an identifier that can be used in the construction of avRules. It can be used to support the definition of collections of avAuthorizations. The collection of avAttribute identifiers in a system must be separate from the domain of avTypes.*

DEFINITION 4. *Given a set T of avTypes, a set C of avClasses, a set A of avAttributes, and a set P of avPermissions, an* **avRule** *is a quintuple* ⟨ ruleType, ruleSource, ruleTarget, ruleClass, ruleAction ⟩, *where:*

ruleType *is either* allow *or* neverallow*;*[2]

ruleSource *is a* pattern*, structured in two parts: (a) a set of positive elements $p_i \in T \cup A$, and (b) an optional set of negative elements $n_i \in T \cup A$;*

ruleTarget *is a* pattern*, with the same structure as the* ruleSource*;*

ruleClass *is a set of avClasses, i.e., each $c_i \in C$, denoting the types of resource that are considered by the avRule;*

ruleAction *is a set of avPermissions, where we assume that each $a_j \in \cap_i c_i.$permissions, i.e., all the elements have to be compatible with all the avClasses specified in the avRule.*

Each avRule can be represented in a textual form, listing the five components following the order above. The textual notation for patterns keeps all the elements within curly braces, preceding the set of negative elements with a "-" character; a colon separates the ruleTarget from the ruleClass.

EXAMPLE 2. *In order to group the common avAuthorizations granted to myapp it is possible to create the avAttribute sdcard and assign it to the sdcard_internal and sdcard_external (through the use of* typeattribute*, defined below). Then, the avAuthorizations defined in Example 1 can*

[2]SELinux also supports the *auditallow* and *dontaudit* rules, which describe the configuration of the auditing services. The model we describe can be easily extended to manage these services.

be derived by the following avRule: ⟨ allow*,* myapp*,* sdcard*,* file*,* read ⟩*.*

The avRules provide a higher-level representation of avAuthorizations. Every *allow* avRule is managed with an expansion of the sets associated with the source, target, class, and action. In general, a cartesian product is computed of all the elements in the positive part. The negative portion of each pattern is used to specify exceptions in the consideration of the positive portion of the pattern.

EXAMPLE 3. *In order to provide myapp the avAuthorizations needed to create and write files and directories labeled with an avType that has the avAttribute sdcard, defined in Example 2, with the exception of the avType sdcard_internal, the required avRule is as follows:* ⟨ myapp*,*
{ sdcard -sdcard_internal }*,* { file dir }*,* { create write } ⟩*.*

An element that has a strong impact on the derivation of the low-level avAuthzPolicy is the definition of the association between avTypes and avAttributes.

DEFINITION 5. *The* typeattribute *statement associates an avType with one or more avAttributes. The syntax of* typeattribute *appears in Table 1. The interpretation is that the avType will be associated with all the privileges that have been granted to the avAttribute.*

DEFINITION 6. *An* **avRulePolicy** *is a set of avRules and* typeattribute *statements.*

The avAuthzPolicy is obtained by a compilation of the avRulePolicy. The compilation is executed by the *checkpolicy* tool, with a sequence of three steps: (a) the typeattribute statements are processed, creating new avRules for every avRule where the avAttribute appears, replacing the avAttribute with the avTypes; (b) all the *allow* rules are expanded, producing a set of avAuthorizations; (c) all the *neverallow* rules are expanded and the policy is checked for the presence of conflicts: if even one avAuthorization produced by the expansion of *neverallow* rules matches an avAuthorization produced by *allow* rules, the compilation stops and an empty policy is produced.

5. REQUIREMENTS

Analyzing the introduction of appPolicyModules in the management of per-app security, we need to consider the different cases that emerge from the combination of the *system policy* and an appPolicyModule. From the model presented above, we note that every avAuthorization defined in an SELinux policy has a *source* avType and a *target* avType. These types may be defined in either the *system policy* or the appPolicyModule. We then have four types of avAuthorization, depending on the origin of the source and target domains. Each configuration is associated with a specific requirement that must be satisfied by appPolicyModules.

Each requirement will be described and formalized using a simple formalization that expresses each requirement as a constraint on the relationship between the system avAuthzPolicy AV, derived from the system avRulePolicy S, and the avAuthzPolicy AV', obtained after the integration of an appPolicyModule M with S. We will show in Section 6 that our proposed language and restrictions for the appPolicyModules satisfy all the requirements. We assume that

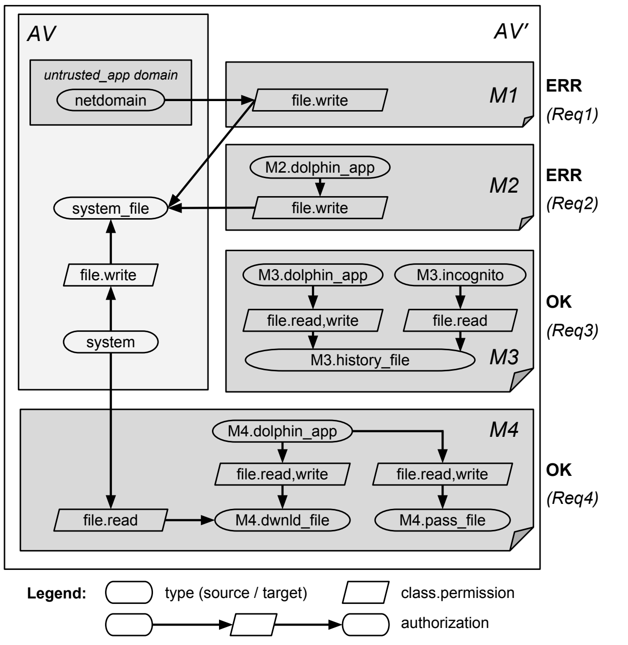


Figure 1: Examples of both compliant and non-compliant modules to illustrate requirements.

there is an avType that describes the domain of safe-to-use resources and actions, called *untrusted_app*, which protects system resources from the abuse of third-party apps (this is the name actually used in the current SEAndroid policy).

An example referring to the *Dolphin Browser* app will also be presented for every requirement, to clarify the impact in the design of the policy. To denote the type of avAuthorization, we use a compact notation where S and A represent respectively the system and appPolicyModule origin of the avType, with this structure: *source* → *target*.

Req1 ($S \to S$), No impact on the system policy: the app must not change the system policy and can only impact on processes and resources associated with the app itself.

An appPolicyModule is intended to extend the system policy and to be managed by the same software modules that manage the system policy. Since third-party apps can not be trusted a priori, it is imperative that the provided appPolicyModule must not be able to have an impact on privileges where source and target are system types.

More formally, AV must be contained into AV' and all the avAuthorizations appearing in $AV' - AV$ have to present as source or target avTypes defined in M (a set represented by notation $M.newAvTypes$).

I.e., $AV \subseteq AV' \;\wedge\; \forall a \in (AV' \setminus AV) \;\to$
$a.source \in M.newAvTypes \vee a.target \in M.newAvTypes$

EXAMPLE 4 (FIGURE 1, M1). *The APM associated with* Dolphin Browser *can specify access privileges only on its own resources, such as its own password database, but must not be able to specify authorizations on system resources. Without this restriction the appPolicyModule could provide* unstrusted_app *write access to the type* platform_app_data_file *and corrupt the system resources, or enhance the privileges of system resources that the app can access, creating unpredictable vulnerabilities. Consider the appPolicyModule* M1 *in Figure 1. The module defines an authorization that modifies the behavior of the system policy because* netdomain *should not have write access to files labeled as* system_file*. Thus* M1 *will not be installed due to the violation of* Req1*.*

Req2 ($A \to S$), No escalation: the app cannot specify a policy that provides to its types more privileges than those available to *untrusted_app*.

New domains declared in an appPolicyModule must always operate within the boundaries defined by the system policy as acceptable for the execution of apps. When a new application is installed, its domain has to be created "under" the *untrusted_app* domain, so the system policy can flexibly define the maximum allowed privileges for third-party apps.

More formally, all the avAuthorizations introduced by the appPolicyModule M that have an avType t belonging to the avTypes defined by M as a source will be contained in the set of avAuthorizations that have the system-defined *untrusted_app* avType as source.

I.e., $\forall\ a' \in (AV' \setminus AV) \mid$
$a'.source \in M.newAvTypes \wedge a'.target \notin M.newAvTypes \rightarrow$
$\exists a \in AV \mid$
$(a.source = untrusted\_app \wedge a'.target = a.target \wedge$
$a'.class = a.class \wedge a'.action = a.action)$

The constraint forces the avAuthzPolicy to assign to all the types introduced by the appPolicyModule a set of authorizations that corresponds to privileges available to the *untrusted_app* avType. Then, each privilege must have the same class and action of a privilege already assigned to *untrusted_app*.

Example 5 (Figure 1, M2). *As highlighted by appPolicyModule* M2 *in Figure 1, appPolicyModules can only request a subset of the privileges granted to the* untrusted_app *domain. The APM* M2 *tries to give* M2.dolphin_app *the privilege of writing files labeled as* system_file*, that is not granted to the* untrusted_app *domain. Thus* M2 *will not be installed due to the violation of* Req2*.*

Req3 ($A \to A$), Flexible internal structure: apps may provide many functionalities and use different services (e.g., geolocalization, social networks). The appPolicyModule has to provide the flexibility of defining multiple domains with different privileges so that the app, according to the functionality in use, may switch to the one that represents the "least privilege" domain needed to accomplish the job, in order to limit potential vulnerabilities deriving from internal flaws.

Greater flexibility derives from the possibility to freely manage privileges for internal types over internal resources, building a MAC model that remains completely under the control of the app.

More formally, there can exist a pair of new avTypes t and t' introduced by M such that in $AV' \setminus AV$ t receives a privilege that t' does not have.

I.e., if $\exists t \in M.newAvTypes \wedge t' \in M.newAvTypes \wedge$
$a \in AV' \wedge a.source = t \not\rightarrow$
$\exists a' \in AV' \mid a'.source = t' \wedge a.target = a'.target \wedge$
$a.class = a'.class \wedge a.action = a'.action$

Example 6 (Figure 1, M3). Dolphin Browser *provides the* anonymous surfing *(*incognito*) mode, which allows the user to surf the web without storing permanently the history and the cookies, and in general aiming at leaving no trace in persistent memory of the navigation session. In order to enhance its security and protect the user even from possible app flaws, the appPolicyModule could specify a switch of context (i.e., it may change the SELinux domain associated with its process) when the user enters the incognito mode. In Figure 1 the APM* M3 *specifies that the domain* M3.dolphin_app *can read and write files labeled as* M3.history_file*, while the domain* M3.incognito*, used during anonymous surfing, drops the privilege of writing the files, preventing the leakage of resources that may leave a trace of the navigation session.*

Req4 ($S \to A$), Protection from external threats: users of mobile devices may unconsciously install malware apps from untrusted sources that, exploiting some security vulnerabilities, could compromise the entire system or other apps (e.g., steal user information). To mitigate the risk, an appPolicyModule should provide a common way to isolate the app's critical resources. The use of MAC support offers protection even against threats coming from the system itself, like a malicious app that abuses root privileges.

The app can protect its resources from other apps, specifying its own types and defining in a flexible way which system components may or may not access the domains introduced by the APM. This requirement depends on the ability of the MAC model to let app types be protected against system-level elements, an aspect that SELinux supports and not available in classical multi-level systems, which assume a rigid hierarchical structure. Indeed, in the SELinux policy model every privilege has to be explicitly authorized and new avTypes are not accessible by system avTypes unless a dedicated rule is introduced in the appPolicyModule.

More formally, the appPolicyModule M can introduce an avAuthorization that gives to an avType introduced by M a privilege that is not necessarily available to a type in the system policy.

I.e., if $\exists\ t \in M.newAvTypes \wedge a \in AV' \wedge\ a.source = t \not\rightarrow$
$\exists a' \in AV \mid a'.source \notin M.newAvTypes \wedge a.target = a'.target \wedge$
$a.class = a'.class \wedge a.action = a'.action.$

Example 7 (Figure 1, M4). *The* Dolphin browser *can grant to the* system *type the privilege to read the* dwnld_file *files, used to label the downloaded files, while it prevents the access to the* pass_file *files used to label the password file.*

There are other environments where SELinux is used, like the Redhat Fedora distribution of Linux, that already supports SELinux modules, but the requirements presented above do not apply to them. The reason is that the trust assumptions are different. The modules used in Redhat Fedora permit to structure the security policy, they are trusted and free to revise in arbitrary ways the system policy. Modules in Android are not trusted and it is mandatory that they cannot be used to introduce vulnerabilities in the system.

Additional requirements, not associated with a formal treatment, have also to be considered.

- Not all the developers have the knowledge or are interested to secure their apps with SELinux, so in order not to impede the development they have to experience the same development and installation process, with no impact on their activities. This requirement will be considered in Section 7.

- In order to facilitate the deployment, the solution has to be compatible with the implementation of SELinux offered by SEAndroid. This is considered in Section 8.

6. POLICY MODULE LANGUAGE

We now present the concrete structure of appPolicyModules. We introduce the subset of the SELinux statements used in their definition and describe the additional statements that will be automatically added to the appPolicyModule by a pre-processor. A critical design requirement is the compatibility with the SELinux implementation available today, which facilitates the adoption of the proposed approach.

Each module presents a head and a body (see right side Figure 2). The head describes all the identifiers that the appPolicyModule reuses from the system policy. This is represented by the *require* statement. In case a name appears that is not known to the system, the compilation fails.

The body of the appPolicyModule can make use of the following SELinux statements: *typebounds*, *type*, *attribute*, *typeattribute*, *allow*, *neverallow*, and *typetransition.* These statements are the only ones that can be used in the definition of the appPolicyModule. The syntax for these statements is succinctly presented in Table 1.

The *typebounds* statement permits to specify that the collection of privileges of the bounded avType has to fall within the boundaries of another avType. The *typebounds* statement will raise an exception when an *allow* rule introduces a privilege for a bounded type in the source that does not match an existing rule for the bounding type.

```
type dolphin_app;
type dolphin_app_incognito;
typebounds untrusted_app dolphin_app,
    dolphin_app_incognito;
allow dolphin_app app_data_file:file
    {read write};
allow dolphin_app_incognito
    app_data_file:file {read};
```

Listing 1: Example of use of *typebounds*. In the system policy there is a rule `allow untrusted_app app_data_file:file {read write};`

The evaluation of compatibility takes into account the presence of other *typebounds* statements in the target, considering as correct the use in the target of an avType that is bounded by the type appearing in the higher-level rule. In the example in Listing 1, the verification by typebounds is satisfied, because both the *allow* rules use in the target an avType that is considered compatible with the *untrusted_app* type. It is useful to emphasize that the *typebounds* statement does not assign the authorizations to the bounded domain, it only sets its upper bound. This is a core principle in our scenario, where policy writers are outside of the trust domain of the core system resources.

The *type* statement permits to introduce new avTypes. To avoid name conflicts between types defined in different modules, the pre-processing adds a prefix that derives from the app name to every identifier (we omit in the examples the representation of this step). If it does not already appear in the module, the pre-processing step will add a *typebounds* statement for every introduced type that will constrain the authorizations referring to types in the system policy to lie within the *untrusted_app* type. The *attribute* statement declares an identifier that can be used to define rules. SELinux policies make extensive use of avAttributes to provide a structure to policies. No constraint needs to be introduced on the definition of new attributes. Attributes produce an effect on the policy when they are used in the *typeattribute* statement, which has been presented above. The pre-processing checks that every *type_id* used in a *typeattribute* statement must be defined inside the appPolicyModule. Without this constraint, a module could violate *Req1* and *Req2*, compromising the system policy and possibly performing an escalation of privileges, by assigning attributes to the *untrusted_app* type. The *allow* and *neverallow* statements permit to create avRules. The pre-processing checks that all the avRules present as a source or target one of the avTypes and avAttributes introduced by the module.

Table 1: Simplified SELinux syntax used in APMs.

Statement	Syntax
attribute	attribute *attribute_id* ';'
type	type *type_id* (',' *attribute_id*)* ';'
typeattribute	typeattribute *type_id attribute_id* (',' *attribute_id*)* ';'
typebounds	typebounds *bounding bounded* (',' *bounded*)* ';'
typetransition	type_transition *type_id type_id*':' '{'*class_id*+'}' *type_id* ';'
allow	allow '{'*pattern*+ ('-'*pattern*)* '}' '{'*pattern*+ ('-'*pattern*)* '}' '{'*class_id*+'}' '{'*perm_id*+'}' ';'
neverallow	neverallow '{'*pattern*+ ('-'*pattern*)* '}' '{'*pattern*+ ('-'*pattern*)* '}' '{'*class_id*+'}' '{'*perm_id*+'}' ';'

The element *pattern*=(*type_id*\|*attribute_id*) is not included in the SELinux statements; we use it here to provide a more readable description of the syntax.

Finally, the *typetransition* statement permits to describe the admissible transitions between avTypes at runtime. We introduce the constraint, checked by the pre-processor, that the avType defined as first parameter has to be an avType defined in the module. The *type_transition* statement is used to perform *object* and *domain* transitions.

- An *object transition* occurs when an object needs to be relabeled (i.e., a file label is changed).
- A *domain transition* occurs when a process with one avType (we call it *transition-startpoint*) switches to another avType (we call it *transition-endpoint*), enacting different avAuthorizations from the original ones. An app could define different domains with limited avAuthorizations and use them when performing specific actions. We note that for a domain transition to succeed, we must grant three different avAuthorizations to the transition-startpoint type.

With respect to object transitions, there is no need to further constrain them, because the process domain must have the corresponding avAuthorizations to be able to create objects with the new label. With respect to domain transitions, when a type transition is authorized and the transition-startpoint type is given the three additional authorizations, the transition-startpoint type is actually able to benefit from all the authorizations that have the transition-endpoint as the source. This is a potential risk in the definition of the policy, because the *typebounds* statement does not extend its evaluation to the consideration of the types that are reachable through type transitions. The current AOSP system policy does not give to *untrusted_app* any type transition privilege, and at the moment there is no danger, but to avoid any risk we enforce the constraint to accept in the appPolicyModule only type transitions that have a transition-endpoint bounded within *untrusted_app*.

6.1 Correctness

We want to show that the appPolicyModules will satisfy the four requirements described in Section 5.

With respect to *Req1 (S→S), No impact on the system policy*, we note that the appPolicyModule statements do not have an impact on the system policy, because all the *allow* and *neverallow* statements have to specify as source or target a new avType, guaranteed to be outside of the system policy.

The correctness with respect to *Req2 (A→S), No escalation* is guaranteed through the use of the *typebounds* statements associated with all the avTypes that appear as source in the *allow* statements. It is to note that the *neverallow* rules do not have to be considered here, because they may only cause the rejection of the appPolicyModule by the compiler, but they cannot lead to the escalation of privileges for the new avTypes. The consideration by the compiler of the *typebounds* statements indeed verifies that each *allow* rule r' in the appPolicyModule that refers to system types has a corresponding *allow* rule r associated with *untrusted_app*.

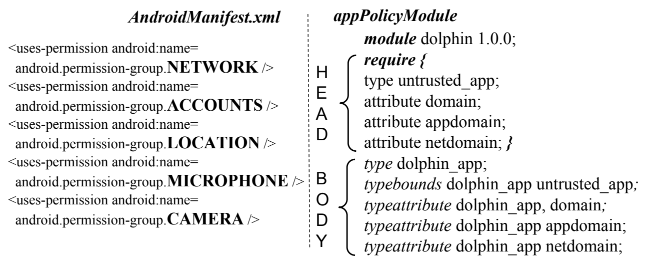
The respect of *Req3* and *Req4* can be demonstrated with a simple example of an appPolicyModule that shows the desired behavior. Requirement *Req3 (A→A), Flexible internal structure* is satisfied by the example in Listing 1, which shows two new avTypes *dolphin_app* and *dolphin_app_incognito*, associated with distinct privileges.

Requirement *Req4 (S→A), Protection from external threats* is supported by the same example: without an explicit rule giving permission, a process associated with *untrusted_app* is not authorized to access files associated with *dolphin_app*.

7. MAPPING ANDROID PERMISSIONS

The introduction of *appPolicyModules* improves the definition and enforcement of the security requirements associated with each app. However, in the approach presented in the previous section, we assumed that the extension to the MAC policy has to be defined by the developer, who knows the service provided by the app and its source code. Due to the size of the community, we can expect that many app developers will either be unfamiliar with the SELinux syntax and semantics, or know SELinux but not want to use it, avoiding the introduction of strict security boundaries to the app beyond those associated with *untrusted_app*. There is also the risk generated by the presence in devices of a variety of versions of the system policy and the need to guarantee that the appPolicyModule is compatible with it.

However, we observe that the app developers that can be expected to be most interested in using the services of the MAC model are expert developers responsible for the construction of critical apps (e.g., apps for secure encrypted communication, or for key management, or for the access to financial and banking services). This community is possibly small, but their role is extremely critical. They can be expected to overcome the obstacles to the use of appPolicyModules. In addition, the deployment of the policy modularity services opens the door to a number of other services. We consider here how it is possible to use them to enforce a stricter model on the management of Android permissions, relying on the automatic generation of appModulePolicies, solving all the issues identified above.



AndroidManifest.xml

```
<uses-permission android:name=
  android.permission-group.NETWORK />
<uses-permission android:name=
  android.permission-group.ACCOUNTS />
<uses-permission android:name=
  android.permission-group.LOCATION />
<uses-permission android:name=
  android.permission-group.MICROPHONE />
<uses-permission android:name=
  android.permission-group.CAMERA />
```

appPolicyModule

```
module dolphin 1.0.0;
require {
type untrusted_app;
attribute domain;
attribute appdomain;
attribute netdomain; }
type dolphin_app;
typebounds dolphin_app untrusted_app;
typeattribute dolphin_app, domain;
typeattribute dolphin_app appdomain;
typeattribute dolphin_app netdomain;
```

Figure 2: Generation of the Dolphin browser appPolicyModule starting from the permissions in the app manifest.

Looking at the workflow to build an app, developers are already familiar with the definition of security requirements in the *AndroidManifest.xml*, through the use of the tag *uses-permission*. In fact, in order to access system resources (e.g., access to the user's current location) the app has to explicitly request the associated Android permissions (e.g., *android.permission-group.LOCATION*), which correspond both to a set of concrete actions at the OS level and to a set of *avPermissions* granted at the SELinux level (e.g., *open*, *read* on files and directories). The system already offers both a high-level representation and a low-level representation of the privileges needed to access a resource, but they are not integrated and what happens, in the absence of policy modularity, is that the app is associated with the *untrusted_app* domain, which is allowed to use all the actions that correspond to the access to all the resources that are invokable by apps, essentially using for protection only Android permissions. The integration of security policies at the Android permission and MAC levels offers a more robust enforcement of the app policy.

This can be realized introducing a mechanism that bridges the gap between different levels, through the analysis of the high-level policy (i.e., the permissions asked by the app within the Android Permission Framework) and the automatic generation of an appPolicyModule that maps those Android permissions to a corresponding collection of SELinux statements. The generator starts from the representation of the app security requirements expressed in the *AndroidManifest.xml*, builds a logical model of the structure of the *appPolicyModule*, and it finally produces the concrete implementation of the *appPolicyModule* and verifies that all the security restrictions are satisfied.

A necessary step in the construction of the mechanism is the identification of a mapping between policies at the distinct levels. The Android Permission Framework contains more than 200 permissions and most of them present a mapping between the Android permission and a dedicated SELinux domain, already specified in the system policy. The current system policy does not cover all the permissions; e.g., the *downloads*, *calendar*, and *media content* resources are associated with the single *platform_app_data_file* type. We expect this aspect to be manageable with a revision of the policy. However, there is a number of Android permissions that can only be partially supported by this mechanism due to current limitations in the security mechanisms provided by internal components (e.g., SQLite).

To summarize, it is already possible to capture most Android permissions in a precise way and some of them with some leeway, leading in all cases to a significant reduction in the size of the MAC domain compared to what would otherwise be associated with an app. We provide in Figure 2 an example of the appPolicyModule that would be generated for the Dolphin browser described in Section 3.1. We note that every app will have to be associated with the *domain* and *appdomain* attributes, which provide all the basic privileges required to let an app execute in the system. The head of the module will then have to introduce the *require* declarations that specify this attribute, together with the *untrusted_app* type and the *netdomain* attribute. The body of the module introduces the *dolphin_app* type, the *typebounds* and all the MAC privileges required to access the network and other resources/services. Due to the current SELinux policy structure the access to the *ACCOUNTS*, *LOCATION MICROPHONE* and *CAMERA* services are mitigated by the *system_server*. The *system_server* is the core of the Android system which manages most of the framework services. The access to the requested services is granted by the *system_server* to the *appdomain* type and through the use of the rule *typeattribute dolphin_app appdomain* the *dolphin_app* type inherits these privileges.

In general, with the availability of appPolicyModules, the system could evolve from a scenario where each app is given at installation time access at the SELinux layer to the whole *untrusted_app* domain, to a scenario where each app is associated with the portion of *untrusted_app* domain that is really needed for its execution, with a better support of the classical "least-privilege" security principle.

8. IMPLEMENTATION

The work done by Smalley et al. in [15] represents the basis for our work. We have introduced a set of extensions in order to enrich the current implementation and manage appPolicyModules. We now provide a description of the challenges to enable the concrete use of appPolicyModules in Android. The system has been implemented with an open source license, extending the current version 4.4.4 of the AOSP (link omitted for the anonymity constraints); adaptation to Android L is planned as soon as it will be released.

The current SELinux implementation for Android spans different levels of the Android stack. At the *Application Framework* level, the *SELinux* class provides access to the centralized Java Native Interface (JNI) bindings for SELinux interaction. The *android_os_SELinux.cpp* file represents the JNI bridge. At the *Libraries* level, the SELinux implementation consists of the *libsepol* and *libselinux* libraries. The former provides an API for the manipulation of SELinux binary policies. The latter provides the APIs to get and set process and file security contexts and to obtain security policy decisions.

The first challenges to the integration of appPolicyModules in Android appeared in the adaptation of the current SELinux libraries and in the addition of the libraries needed to build, link and check the appPolicyModules. These libraries are part of the full SELinux environment and are included in the major Linux distributions, but the activation

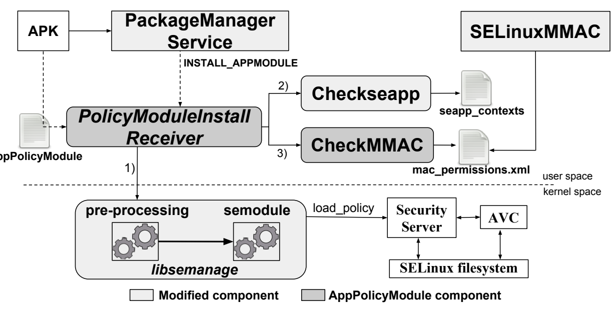


Figure 3: appPolicyModule Architecture.

of policy modules in SELinux for Android required more than a simple cross-compilation.

8.1 Changes to SELinux

The work we did at the SELinux level can be structured into four major activities.

First, the *libselinux* library was modified with the introduction of additional features needed by the *libsemanage* library, such as the *selinux-config.c* module. We modified the *checkpolicy* tool in order to build automatically the binary policy at version 26 (standard SELinux implementations support version 29 of the binary policy). In the current SEAndroid implementation the binary policy version range is between 15 and 26. This constraint is enforced by the *load_policy* method when a policy reload is triggered.

Second, the *libsemanage* library, which provides APIs for the manipulation of SELinux binary policies and binary policy modules, was adapted to fulfill the new requirements. Due to the differences in scenario, architecture and requirements, some functions, such as the *genhomedircon* service, were disabled. The *genhomedircon* service is used to generate file context configuration entries for user home directories based on their default roles and is run when building the policy. However, in Android, though there is the possibility to create several users for a single device, they do not have a *home* directory.

Third, the source code of the *semodule* executable was extended in order to correctly interact with the modified version of the *libsemanage* library. The *semodule* tool is used to manage SELinux policy modules, including installation, upgrade, listing and removal of modules. The *semodule* tool may also be used to force a rebuild of the policy from the module repository and/or to force a reload of the policy without performing any other transaction.

Figure 3 shows an abstract representation of the complete architecture introduced in order to manage the appPolicyModules. Fourth, in addition to the modifications on the set of SELinux libraries, to meet the requirements introduced in Section 2 a pre-processing phase was introduced. This phase supports the creation of constraints introduced in Section 4. Thanks to the modularity provided by SELinux, we were able to implement the pre-processing phase reusing several SELinux components.

8.2 Changes to Android

The second set of challenges concerns the app installation process, which starts from the APK file that contains the app. The *PackageManagerService* class provides the APIs that actually manage app installation, uninstallation, and update. The *PackageManagerService* component provides the functions to parse the *APK* file and to assign the SELinux label to the app. The label is retrieved by the *SELinuxMMAC* class from the *mac_permissions.xml* file. The file maps the app certificate to a SELinux label. In the current AOSP version, all third-party apps are assigned to the default stanza of the *mac_permissions.xml* file, regardless of their certificate. To address this limitation and assign to the app the right SELinux type, we introduced a new install service, named *PolicyModuleInstallReceiver*. This service is triggered by an Intent and manages the installation workflow of an appPolicyModule. The workflow is structured as follows:

1. trigger the installation of the policy module, update the SELinux policy and check its correctness;
2. update the *seapp_contexts* file used to label app processes and app package directories;
3. update the *mac_permissions.xml* file used by *SELinuxMMAC* to retrieve the type to assign to the app. This file is used in conjunction with *seapp_contexts*.

8.2.1 Update SELinux policy

The use of the *PolicyModuleInstallReceiver* service requires to broadcast an *intent.action.INSTALL_APPMODULE* intent. When the service receives the intent it performs a JNI call to the *libsemanage* library, which validates (i.e., preprocessing phase), links, expands the module (i.e., *semodule* tool) and triggers the reload of the binary policy.

8.2.2 Update seapp_contexts

In order to meet the requirements Req3 and Req4 described in Section 5 the app processes and the app package directories have to be labeled accordingly to the appPolicyModule. In the current AOSP implementation the security context assigned to app processes, respectively app package directories, is retrieved from the seapp_context file by the *selinux_android_setcontext*, respectively *selinux_android_setfilecon2*.

In order to manage the addition of new entries in the *seapp_contexts* file the *checkseapp* utility was extended. Currently, *checkseapp* is used during the AOSP build process to validate the *seapp_contexts* file against policy. The extension permits to dynamically manage the addition/removal of an entry in the *seapp_contexts* according to the domains defined in the appPolicyModule.

To allow AOSP components such as *Zygote* to spawn applications in the correct domain, an update of the *mac_permissions.xml* is needed.

8.2.3 Update mac_permissions.xml

This file is used to configure the install MMAC policy. More specifically this file is used in conjunction with the *seapp_contexts* file in order to determine the *seinfo* label to assign to the app. The *seinfo* value is subsequently used to determine the SELinux security context for the app process and its */data/data* directory based on the *seapp_contexts* configuration.

The information needed to build a stanza for the *mac_permissions.xml* file are (i) the app's X.509 certificate and (ii) the *seinfo* label. The stanza is built on the fly by the *checkMMAC* Java class retrieving the information directly from the parsed *apk* and the entry added to the *seapp_contexts* file. After stanza creation, an update is triggered to insert and refresh the whole *mac_permissions.xml* file. This is needed in order to let *SELinuxMMAC* retrieve the right *seinfo* label for the new app.

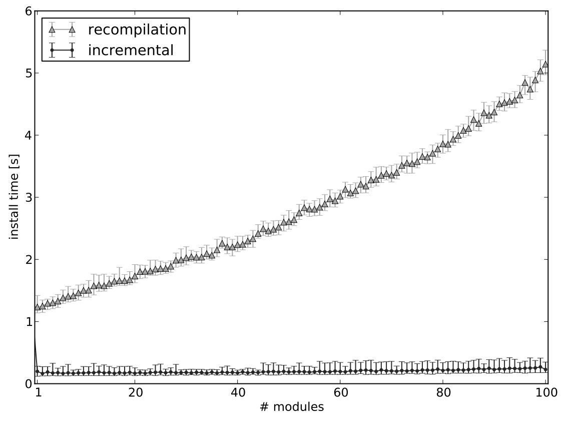


Figure 4: Installation time: comparison between the full recompilation and the incremental approach.

8.3 Performance

As it was clearly expressed in the design of SEAndroid [15], it is necessary to have a minimal overhead in terms of performance, both at app installation time and during regular system runtime. We executed a series of experiments for the evaluation of the performance impact of the techniques presented in this paper. Experiments have been run on a Nexus 7 2013 aosp_flo_userdebug 4.4.2_r2. The Android runtime used was ART.

8.3.1 Installation time

We evaluated the performance overhead of our approach at app installation time, due to the fact that the process to install an app was extended in order to manage appPolicyModules. Two different approaches were developed; the first one is consistent with the approach used for SELinux in Fedora. When a new module is ready to be installed, the *libsemanage* tool creates a new version of the policy and re-installs all the old modules plus the new one. If the new policy passes the checking step, then the new policy is stored into the system, otherwise a rollback occurs. This approach introduces a non negligible overhead, because it requires to reconsider the whole policy every time a new module is added to the system. This option is natural for SELinux in Fedora, because the modules are created by trusted entities and they are free to modify the system policy, with a number of types in the existing binary representation that may have to be updated. As we have already discussed, the requirements associated with the use of appPolicyModules in Android are different, because in our scenario an appPolicyModule cannot modify the system policy (*Req1* in Section 5). In addition to the security benefit, this requirement also leads to a simplification of the management of policy modules, because the re-installation of the system policy and all the other modules is not needed. This permits to provide an "incremental" solution, with a significant reduction in the appPolicyModule installation time.

The graph in Figure 4 describes the time observed in a scenario where the current system policy has been extended with 100 modules, adding the modules one by one. The tests were run 100 times. Each element in the graph describes the range of measured values and the average. The observations in the upper part of the graph show that as the number of modules increases, the re-compilation approach shows a significant increase in the compilation time, due to the fact that at each step the policy becomes larger and its full recompilation more expensive. The incremental approach instead shows a constant response time, with no observable impact deriving from the increase in the size of the overall policy. The average installation time for the incremental approach, which is the one to use, is near to 0.2 s, compatible with the requirements of real systems.

8.3.2 Runtime

We evaluated the performance overhead of our approach at runtime, considering two scenarios with different binary policy sizes.

Table 2: Binary policies used in the tests.

policy	#rules	size
sepolicy	1319	73KB
sepolicy +1000 APMs	35319	631KB

For runtime analysis we used two well known benchmark apps: (i) AnTuTu [1] by AnTuTu Labs and (ii) Benchmark by Softweg [16]. Under both benchmarks, we ensured that the same number of apps/services were loaded and running.

Table 3: AntuTu Benchmark (100 iterations), higher values are better

	sepolicy	sepolicy +1000 APMs
svm	1130.467	1132.867
smt	3334.533	3341.400
database	630.600	631.333
sram	1534.840	1538.533
float	1938.600	1939.200
snand	1159.320	1159.400
memory	1121.280	1120.800
integer	2285.933	2284.133

AnTuTu Benchmark is a popular Android utility for benchmarking devices. As it was explained in [15], the overhead introduced by SELinux is very limited and it only affects *sdwrite*, *sdread* and *database I/O* tests. The tests performed by Smalley et al. take into account a "static" policy. In our scenario the policy size is not static, but it changes at each installation and can potentially become quite large. However, experimental results highlight how the policy size does not affect the system performance. Table 3 shows the results of the benchmark. The impact of the larger policy is not detectable by the experiments.

Table 4 shows the results provided by *Benchmark* app developed by Softweg. Similarly to the results obtained by AnTuTu, SELinux does not affect CPU and graphics scores. For the filesystem and sdcard tests, the overhead introduced by the increased size of the policy is negligible. As highlighted by the *create and delete* tests, the time taken to create or delete 1000 empty files increased by less than 1 percent. As explained by Smalley et al. [15] the *create and delete* tests can be viewed as a worst case, since the overhead of managing the *security context* is not amortized over any real usage of the file.

Table 4: Softweg Benchmark (100 iterations): for the Total File System score, higher values are better

	sepolicy	sepolicy +1000 APMs
Create 250 empty files	1.222 s	1.230 s
Create 1000 empty files	0.302 s	0.303 s
Delete 250 empty files	0.351 s	0.351 s
Delete 1000 empty files	0.130 s	0.130 s
Total file system score	342.835	341.158

9. RELATED WORK

In the past few years a strong interest has been dedicated to the investigation of Android security. Several solutions have been proposed to increase the security of the system and to protect the apps and system components from a variety of threats. The central role of the proposal by Smalley et al. [15] has already been discussed. We summarize here other important contributions in the area.

TaintDroid, proposed in [8] by Enck et al., provides functions to detect the unauthorized leakage of sensitive data. *TaintDroid* uses dynamic taint analysis (i.e., taint tracking) to monitor the exchange of sensitive information among third-party apps. While this solution try to identify the information leakage, our proposal goes one step further impeding the leakage at the SELinux level.

Other solutions, such as *FlaskDroid* [6], *TrustDroid* [5] and *XManDroid* [4] show greater similarity to our work.

FlaskDroid [6] is a security architecture for the Android OS that instantiates different security solutions. It is inspired by the concepts of the *Flask* architecture and is based on SEAndroid. *FlaskDroid* provides mandatory access control on both Android's middleware and kernel layers. This represents an enhancement in terms of the isolation that is provided between separate components, but the two MAC levels are not coordinated and largely use *booleans* in the SELinux policy. In the current SEAndroid implementation, the use of booleans inside the policy is strongly discouraged, for two main reasons: (i) it could introduce compatibility problems, and (ii) it could undermine the default security goals being enforced via SELinux in AOSP itself. Compared to our proposal, the focus of *Flaskdroid* is the security of system modules and the security of third-party apps is not supported. *Flaskdroid* does not permit to dynamically add policy modules without a recompilation of the entire policy.

TrustDroid [5] and *XManDroid* [4] provide mandatory access control at both the middleware layer and at the kernel layer. At the kernel layer, they rely upon *TOMOYO* Linux [9], a path-based MAC framework. *TOMOYO* supports policy updates at runtime, but the security model of SELinux is more flexible and supports richer policies.

RootGuard [14] is an enhanced root-management system which monitors system calls, to detect the abnormal behavior of apps (i.e., malware) with root privileges. It is composed by three components (i) *SuperuserEx*, (ii) *Policy storage database*, and (iii) *Kernel module*, which span the different levels of the Android architecture. The *SuperuserEx* is built on top of the open source Superuser app, the *Policy storage database* is used to store the *RootGuard* policy and the *Kernel module* introduces a set of *hooks* in order to intercept system calls. This implementation is similar to the one used by SELinux, but all the SELinux code is already in the mainline Linux kernel and provides a more robust solution.

10. CONCLUSIONS

Security is correctly perceived, both by technical experts and customers, as a crucial property of mobile operating systems. The integration of SELinux into Android is a significant step toward the realization of more robust and more flexible security services. The attention that has been dedicated in the SEAndroid initiative toward the protection of system components is understandable and consistent with the high priority associated with the protection of core privileged resources. Our approach is the natural extension of that work, which demonstrated a successful deployment, toward a more detailed consideration of the presence of apps.

The paper shows that the potential for the application of policy modules associated with each app is quite extensive, supporting scenarios where developers define their own app policy, and scenarios where policies are automatically generated to improve the enforcement of privileges and the isolation of apps. The extensive level of reuse of SELinux constructs that characterizes the language for the appPolicyModules demonstrates the flexibility of SELinux and facilitates the deployment of the proposed solution. An analysis of the evolution of the official SEAndroid project confirms that appPolicyModules identify a concrete need and that Android is evolving in this direction.

11. ACKNOWLEDGEMENTS

This work was partially supported by a Google Research Award (winter 2014), by the Italian Ministry of Research within the PRIN project "GenData 2020" and by the EC within the 7FP and H2020, respectively, under grant agreements 257129 and 644579.

12. REFERENCES

[1] AnTuTu labs. AnTuTu Benchmark. `https://play.google.com/store/apps/details?id=com.antutu.ABenchMark` .

[2] M. Arrigoni Neri, M. Guarnieri, E. Magri, S. Mutti, and S. Paraboschi. Conflict Detection in Security Policies using Semantic Web Technology. In *Proc. of IEEE ESTEL - Security Track*, 2012.

[3] M. Balanza, K. Alintanahin, O. Abendan, J. Dizon, and B. Caraig. Droiddreamlight lurks behind legitimate android apps. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 73–78. IEEE, 2011.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, volume 17, pages 18–25, 2012.

[5] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. 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.

[6] 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, 2013.

[7] Dolphin Browser. Dolphin Browser for Android. `https://play.google.com/store/apps/details?id=mobi.mgeek.TunnyBrowser` .

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

[9] T. Harada, T. Horie, and K. Tanaka. Task oriented management obviates your onus on linux. In *Linux Conference*, volume 3, 2004.

[10] J. Lepreau, R. Spencer, S. Smalley, P. Loscocco, M. Hibler, and D. Andersen. The flask security architecture: System support for diverse security policies, 2006.

[11] A. Ludwig. Android - practical security from the ground up, October 2013. `http://goo.gl/z0RIwu` .

[12] F. Mayer, K. MacMillan, and D. Caplan. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, NJ, USA, 2006.

[13] C. Mulliner, W. Robertson, and E. Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '14, pages 459–470, New York, NY, USA, 2014. ACM.

[14] Y. Shao, X. Luo, and C. Qian. Rootguard: Protecting rooted android phones. *Computer*, 47(6):32–40, 2014.

[15] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Network and Distributed System Security Symposium (NDSS 13)*, 2013.

[16] Softweg. Benchmark. `https://play.google.com/store/apps/details?id=softweg.hw.performance` .

[17] Titanium Track. Titanium Backup. `https://play.google.com/store/apps/details?id=com.keramidas.TitaniumBackup` .

[18] C. Yang, V. Yegneswaran, P. Porras, and G. Gu. Detecting money-stealing apps in alternative android markets. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 1034–1036, New York, NY, USA, 2012. ACM.

[19] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.