SEApp: Bringing Mandatory Access Control to Android Apps

Matthew Rossi, Dario Facchinetti, and Enrico Bacis, *Università degli Studi di Bergamo;* Marco Rosa, *SAP Security Research;* Stefano Paraboschi, *Università degli Studi di Bergamo*

https://www.usenix.org/conference/usenixsecurity21/presentation/rossi

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

SEApp: Bringing Mandatory Access Control to Android Apps

Matthew Rossi
Università degli Studi di Bergamo
matthew.rossi@unibg.it

Dario Facchinetti
Università degli Studi di Bergamo
dario.facchinetti@unibg.it

Enrico Bacis*
Università degli Studi di Bergamo
enrico.bacis@unibg.it

Marco Rosa
SAP Security Research
marco.rosa@sap.com

Stefano Paraboschi
Università degli Studi di Bergamo
parabosc@unibg.it

Abstract

Mandatory Access Control (MAC) has provided a great contribution to the improvement of the security of modern operating systems. A clear demonstration is represented by Android, which has progressively assigned a greater role to SELinux since its introduction in 2013. These benefits have been mostly dedicated to the protection of system components against the behavior of apps and no control is offered to app developers on the use of MAC. Our solution overcomes this limitation, giving developers the power to define ad-hoc MAC policies for their apps, supporting the internal compartmentalization of app components.

This is a natural evolution of the security mechanisms already available in Android, but its realization requires to consider that (i) the security of system components must be maintained, (ii) the solution must be usable by developers, and (iii) the performance impact should be limited. Our proposal meets these three requirements. The proposal is supported by an open-source implementation.

1 Introduction

Security in operating systems has greatly evolved and has been able to address many of the threats originating by an extensive and varied collection of adversaries.

The mitigation of security threats is particularly important for *mobile operating systems*, due to their wide deployment and the confidential information they hold.

Both Android and iOS have seen significant investments toward the realization of advanced security techniques, which have led to a great increase in the level of protection offered to users [58]. The strength of security and the value of protected resources is testified, for instance, by the payouts associated with working exploits in markets like Zerodium [72], where the payouts for mobile operating systems are the highest[1].

A peculiar threat that characterizes mobile operating systems is the need to balance on one side the high sensitivity of the information, and on the other hand the need for users to install into the system a large number of applications (called simply *apps* in this domain) often produced by unknown developers, which may hide malicious functions. A first level of protection is offered, both in iOS and Android, by a preliminary screening of apps before they are made available on the platform market [2] or installed to a device, but this approach cannot provide a strong guarantee. Security mechanisms internal to the operating system are needed in order to constrain the apps to only operate within the boundaries specified by the device owner at installation time.

The approach used in the design of mobile operating systems considers as the first requirement the protection of system resources. Focusing on Android, which is open source and more accessible to researchers, we notice a significant evolution in its internal security architecture. This architecture is quite rich and consists of many security measures [44, 58]. In this environment, we specifically look at the role of SELinux. SELinux implements the *Mandatory Access Control* (MAC) mechanism, which relies on a system-level policy to declare the operations that a process can execute over a resource based on the security labels associated with them. Compared to classical *Discretionary Access Control* (DAC), still used in Android in an extensive way, MAC is more rigid and provides stronger guarantees against unwanted behaviors. When SELinux was introduced into Android 4.3 in 2013 (see Figure 1), it used a limited set of system domains and it was mainly aimed at separating system resources from user apps. In the next releases, the configuration of SELinux has progressively become more complex, with a growing set of domains isolating different services and resources, so that a bug or vulnerability in some system component does not lead to a direct compromise of the whole system.

The introduction of SELinux into Android has been a clear success. Unfortunately, the stronger protection benefits do not extend to regular apps which are assigned with a single domain named `untrusted_app`. Since Android 9, isolation

*now at Google

[1] At the time of writing, US$2.5M and US$2M are paid for a zero click solution able to subvert the security of Androd and iOS, respectively.

of apps has increased with the use of categories, which guarantees that distinct apps operate on separate security contexts. Our proposal, SEApp, builds upon the observation that giving app developers the ability to apply MAC to the internal structure of the app would provide more robust protection against other apps and internal vulnerabilities.

2 Android security for apps

One of the major requirements considered in the design of mobile operating systems is the need to constrain the ability of apps to manipulate the execution environment. Apps may hide functions that are meant to gain system privileges or capture valuable information from other apps. Compared to classical desktop operating systems, there is greater reliance on the use of apps to access resources or get services, with more attention paid to limit the ability of apps to operate in the system. Advancements in this context can have an impact on how security for applications is managed in other domains [1].

The basic principle adopted to manage the threat introduced by apps is the design of a *sandbox*, a restricted environment for app execution, where anomalous actions by the app are not able to access resources beyond what has been authorized at app installation time. The sandbox can be considered a realization of the "least privilege" security principle.

The construction of the app sandbox is based on three access control mechanisms: Android permissions [14, 44, 45], Discretionary Access Control (DAC) [38], and Mandatory Access Control (MAC) [63]; each of them roughly aligning with how users, developers, and the platform grant consent, respectively.

Android permissions restrict access to sensitive data and services. In file `AndroidManifest.xml` [16], each app statically lists the Android permissions needed to fully operate. Not all of them may be granted; depending on the threat they pose from a security and privacy standpoint, they may be granted as part of the installation procedure, or prompted to the user when the app needs them.

DAC restricts access to resources based on user and group identity. By assigning each application a unique UNIX user ID (UID) and a dedicated directory, Android isolates apps from each other and from the system. However, UID sandboxing has a number of shortcomings. As an example, processes running as root are not subject to these restrictions. For this reason, when such a process is misbehaving, for instance due to a bug, it can access private app data files. DAC discretionality itself is a problem. Indeed, as apps and system processes could override safe defaults, they are more susceptible to dangerous behavior, such as leaking files or data across security boundaries via IPC or fork/exec. Despite its deficiencies, UID sandboxing is still the primary enforcement mechanism that separates apps from each other, establishing the foundation upon which further sandbox restrictions have been built.

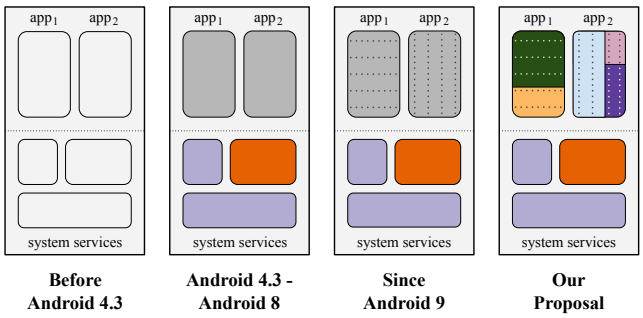


Figure 1: Evolution of the MAC policy in Android. Before 4.3, MAC was not used. Starting with 4.3, MAC protects system components. Since 9, categories offer rigid MAC protection for apps. Our proposal offers flexible MAC protection to apps.

MAC dictates which actions are allowed based on the security policy defined by the system. Specifically, only actions explicitly granted by the policy are permitted. To decide whether to permit or deny an action, a set of policy rules concerning the *security contexts* (i.e., collections of security labels that classify resources) of the involved parties is evaluated.

In Android, MAC is implemented using SEAndroid, a set of kernel modifications part of the Linux Security Module (LSM) framework [70]. Since its first introduction with the Security Enhanced Android (SEAndroid) project [65], SELinux has been extensively applied to protect system components. Initially, it was used to assert the security model requirements during compatibility testing, then its usage grew further at each release. In the current version Android 11, SELinux is also used to isolate the rendering of untrusted web content (by the `isolated_app` domain), to restrict *ioctl* system calls [56], thus limiting the reachability of potential kernel vulnerabilities, and to support multi-user separation and app sandboxing with SELinux categories. This last aspect permits to enforce app separation both at DAC and MAC. Android dynamically assigns categories to apps during app installation, so that: (i) an app running on behalf of a user cannot read or write files created by the same app on behalf of another user (since Android 6 [9]); and, (ii) an app cannot read or write files created by another app (since Android 9 [11]). Before Android 9, this separation was only enforced at DAC level. This overlap of security measures is of extreme relevance to the enforcement of the Android Security Model and our proposal moves in the same direction. To bypass these protections, a process should be granted root permissions, `DAC_OVERRIDE` or `DAC_READ_SEARCH`, and run as SELinux `mlstrustedsubject`; only a few critical system services run in this configuration.

Android restricts the SELinux implementation to the policy enforcement, ignoring most policy management functions. The motivation is that the system policy only changes between releases, therefore support to runtime changes is not needed.

3 Motivation

As discussed above, SELinux and the MAC support have been a crucial factor in the realization of a secure design and the construction of a robust app sandbox. A limitation of the current design is that this is the only way that apps can benefit from MAC support. There is currently no option to let the app developer control the use of the MAC level, as only platform, vendor, ODM and OEM developers are allowed to introduce new policy segments [24]. Our solution overcomes this limitation, giving the application developer the power to specify new SELinux types and associated permissions.

3.1 Use cases

We envision several scenarios that justify the use of SEApp. Many of them have been previously considered by researchers as motivations for the introduction in Android of dedicated components [33, 41, 55].

In this Section we give a tour of SEApp capabilities using a showcase app[2]. The architecture of the showcase app is shown in Figure 2. Our description is based on three use cases: fine-granularity in access to files, fine-granularity in access to services, and isolation of vulnerability prone components. Each of the use cases emphasizes the intra-app security features introduced by SEApp. A dedicated description, along with policy files that show concretely how to enforce these use cases, appears in the Appendix; we provide there a technical demonstration of how SEApp can provide protection against a number of common security problems in Android apps [51] that were implemented in the showcase app.

3.1.1 Fine-granularity in access to files

Android apps can collect data from multiple sources, and the system provides many options to store it. The default one is *Internal Storage*: a filesystem region, located at `/data/data/packageName`, reserved to each package. Its content is available to all app's internal components and inaccessible to any other app. Since data can be extremely sensitive, the developer may be interested in restricting its visibility to only some internal components, labeling sensitive and non-sensitive data with distinct SELinux types (use case 1). Yet, in the current Android security model, apps do not have the option to assign distinct MAC labels to different resources, as all internal files are labeled `app_data_file`. SEApp allows the developer to introduce dedicated types, and to organize the app's structure with a separation between components managing non-sensitive data and those requiring access to sensitive data. The sensitive components will be associated with a more stringent MAC domain. Figure 2 shows an example in which the *confidential* files are made accessible to *:core_logic* processes and inaccessible to any other process.

[2]The showcase app is available in the SEApp repository along with the set of modifications to the AOSP.

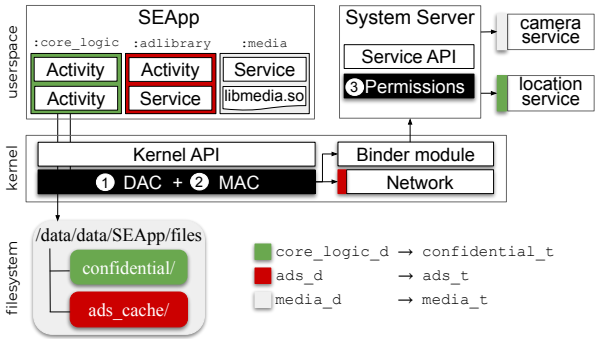


Figure 2: Security Enhanced App

In Appendix A.1 we give a demonstration of how *confidential* files are made inaccessible to non-confidential components in the presence of a path traversal vulnerability.

3.1.2 Fine-granularity in access to services

Often developers introduce into their applications code coming from external sources, which they do not fully trust [40, 46, 61]. For instance, a common need of app developers is to get revenue from their apps and a simple approach is to include an Ad delivery library within the app. The library is a relatively complex piece of code, with local computation necessities and the need to manage a dialogue with remote servers. The app developer is clearly interested in supporting the execution of the library, but may want to have guarantees that the library cannot abuse the access privileges granted by the user to the whole application sandbox (use case 2). A common concern is preventing access to system services such as *location*. These requirements can be managed by SEApp with the definition of a separate MAC domain for the library. The process managing the delivery of Ads will be associated with this domain, which will provide only the necessary privileges to access the dedicated resources needed for the library execution. SELinux will then guarantee the confinement of the library, preventing access to the location service even if the `ACCESS_FINE_LOCATION` permission is granted to the app. Figure 2 shows an example in which the *:adlibrary* process is granted access to the network but is prevented from accessing `location service`.

In Appendix A.2 we give a demonstration of how the showcase app can support the execution of the Unity Ads [69] framework with a dedicated SELinux domain. We also describe in detail how SEApp prevents a malicious component, which was deliberately injected by us into the library process, to capture the device location.

3.1.3 Isolation of vulnerability prone components

App developers often have to consider that the input provided to the app can come from untrusted sources. A typical example is the rendering of complex Javascript code performed by WebView. The solution currently offered by Android is to execute these potentially dangerous actions within a sandbox using *isolatedprocess*, i.e., a special process that is isolated from the rest of the system and has no permissions of its own [6]. It runs under a dedicated UID and SELinux domain, and it can only interact with a restricted number of services [8].

A common need of app developers is to take advantage of complex media or processing libraries, components that are not considered malicious, but due to their size and complexity are more likely to have security bugs. The developer is then interested in isolating these potentially vulnerable components (use case 3). *Isolatedprocess* offers a high protection level in Android, however, its use imposes several restrictions on the developers. For instance, *isolatedprocess* cannot perform many of the core Android IPC functions, and the only way to interact with it is through the bound service API [7]. Also, *isolatedprocess* can only access already open app files received over Binder. Another shortcoming is that each invocation of an *isolatedprocess* requires the creation of a new process. If a series of requests are made by the app, the performance impact can be significant. SEApp offers an easier way to do this compared to *isolatedprocess*, as it permits to assign a domain to the process in which the component is executed, and then configure the required permissions at MAC level. In terms of performance, the management of multiple requests does not require the system to activate a new process with a new UID and a dedicated SELinux category. Figure 2 shows how to confine the *:media* component.

In Appendix A.3 we give a demonstration of how the showcase app can support the execution of media components relying on a native library in a dedicated process. We also describe how the developer can leverage SEApp to prevent the code of the library from the execution of unwanted or unintended operations, like opening a network connection.

3.2 Modular app compartmentalization

The motivations presented above become more frequent as apps increase their size and complexity, and several important apps see a continuous increase in these parameters. For instance, Facebook Messenger version 285 contains more than 500 components and WhatsApp Messenger version 2.20 more than 300. This increase in size and the need to manage it is testified by the development of App Bundles [4], Android's new, official publishing format that offers a more efficient way to build and release modular applications.

In these large and modular apps, developers find it difficult to fully control which components of an app are using sensitive data[3]. The availability of a solution such as SEApp can greatly reduce such risk. A better compartmentalization can reduce the impact of internal vulnerabilities in modular apps, since each module can be associated with a dedicated policy fragment. From a security and software engineering standpoint, SEApp permits to separate the activities of security policy maintenance and development of new features.

3.3 Compatibility with Android design

Looking at the evolution of Android, it is clear that our proposal is consistent with the evolution of the operating system and the desire of its designers to let app developers have access to an extensive and flexible collection of security tools. The major obstacles, as perceived by OS developers, on offering to app developers the use of MAC services are: weakening of the protection of system components; performance impact; usability by app developers. The work we did solves these concerns: our approach guarantees that app policies do not have an impact on the system policy (Section 4.3); the app policy can be specified declaratively and attention has been paid to let developers adopt the approach in a convenient way (Section 5.2); and, experiments demonstrate the acceptable performance impact, with a quite limited overhead at app installation time, and a negligible runtime impact (Section 7).

3.4 Compatibility with other proposals

As presented in Section 3.1, SEApp by itself provides protection against a broad spectrum of attacks (see Appendix), but its merit does not end there. As multiple literature proposals (e.g., [35, 55, 71]) build upon process isolation and use it to accomplish separation of privileges at the application layer, SEApp could be used as building block to enforce such restrictions at the MAC layer too, enabling defense in depth. Moreover, SEApp could also work in conjunction with other solutions that work at MAC level such as *FlaskDroid* [37], to benefit of its Userspace Object Managers (USOMs) coverage of the Android system services and provide finer granularity in access to services.

4 SEApp policy language

To support the use cases presented in Section 3, we want the developer to have control of the SELinux security context of subjects and objects related to her security enhanced app. To each of them is assigned a *type* (also called *domain* when it labels processes). As types directly relate to groups of permissions, the evaluation of security contexts is the foundation of each security decision. Since apps may offer many complex functions, the policy language has to provide the flexibility of

[3] The topic was explicitly considered in [30], an interview with Android's VP of Engineering.

Table 1: Application policy module CIL syntax

Policy module syntax	
blockStmt	→ (`block` *blockId cilStmt*∗)
cilStmt	→ *typeStmt* | *typeAttrStmt* | *typeAttrSetStmt* | *typeBoundsStmt* | *typeTransStmt* | *macroStmt* | *allowStmt*
typeStmt	→ (`type` *typeId*)
typeAttrStmt	→ (`typeattribute` *typeAttrId*)
typeAttrSetStmt	→ (`typeattributeset` *typeAttrId* (⟨*typeId* | *typaAttrId*⟩+))
typeBoundsStmt	→ (`typebounds` *parentTypeId childTypeId*)
typeTransStmt	→ (`typetransition` *sourceTypeId targetTypeId classId* [*objectName*] *defaultTypeId*)
macroStmt	→ (`call` *macroId* (*typeId*))
allowStmt	→ (`allow` ⟨*sourceTypeId* | *sourceTypeAttrId*⟩ ⟨*targetTypeId* | *targetTypeAttrId* | `self`⟩ *classPermissionId*+)

defining multiple domains with distinct privileges so that the app, according to the task it has to do, may switch to the least privileged domain needed to accomplish the job.

The app policy is specified in a module, provided by the app to describe its own types. The policy module is processed at app installation time by a component of the system, called *SEApp Policy Parser*, responsible to verify that the policy is correct and does not introduce vulnerabilities into the system. The addition of a policy module is managed by combining the new module with the platform policy and the previous installed ones, producing after policy compilation a single binary representation of the global policy.

In this section we provide a description of the SEApp policy language and the restrictions each module is subject to. Policy configuration is detailed in Section 5, while policy compilation and runtime support are discussed in Section 6.

4.1 Choice of policy language

SEAndroid supports two languages for policies, Type Enforcement (TE) [67] and Common Intermediate Language (CIL) [57]. TE was the language available in the early implementations of SELinux, while CIL was later introduced to offer an easy to parse syntax that avoids the pervasive use of general purpose macro processors (e.g., M4 [48]). Another aspect that differentiates them is that, in Android, TE representations are internally converted into CIL before being compiled into the SELinux binary policy. To avoid the additional translation step being performed at each policy module installation, we decided to use CIL over TE.

4.2 Definition of types and type-attributes

CIL offers a multitude of commands to define a policy, but only a subset has been selected for the definition of an app policy module. This was done to control the impact of the policy module on the system and it may, as a side effect, facilitate the work of the app developer writing the policy.

The syntax is described in Table 1. To declare a *type*, the `type` statement can be used. This permits to declare the types involved in an access vector (AV) rule, which grants to a source type a list of permissible actions over a target type. AV rules are defined through the `allow` statement.

When writing a policy, there is frequently the need to assign the same set of authorizations to multiple types. To avoid the repetition of multiple `allow` declarations, it is convenient to refer to multiple types using a single entity, the *type-attribute*. Using the `typeattributeset` statement we associate with a `typeattribute` a set of types and type-attributes. Each type-attribute essentially represents the set of types that is produced by the (possibly multi-step) expansion of its definition. The semantics is that each of the types that directly or indirectly (using type-attributes) appears as the source of an allow rule will be authorized to operate with the specified permission on each of the types directly or indirectly appearing as the target. This improves the conciseness and readability of the policy.

After defining the domains with the least group of permissions necessary to fulfill the task, the developer can also configure the domain transitions using the `typetransition` statement. By doing so, it is possible to ensure that important native processes run in dedicated domains with limited privileges, leading to intra-app compartmentalization.

4.3 Policy constraints

The introduction of dedicated modules for apps raises the need to carefully consider the integration of apps and system policies. The first requirement is that an app policy must not change the system policy and can only have an impact on processes and resources associated with the app itself. To preserve the overall consistency of the SELinux policy, each policy module must respect some constraints. Since Android supports the side-loading of apps [3], we cannot rely on app markets to verify app policies. Therefore, the enforcement of constraints is done on the device, by both the SEApp Policy Parser and the SELinux environment. If any of these components raises an exception, during the verification or compilation of the policy, app installation is stopped.

To ensure that policy modules do not interfere with the system policy and among each other, a first necessity is that

policy modules are wrapped in a unique namespace obtained from the package name. This is done through the `block` CIL statement, which prevents the definition of the same SELinux type twice, as the resulting global identifier is formed by the concatenation of the namespace and the local type identifier. Also, the use of a namespace specific for the policy module permits to discriminate between local types or type-attributes T_A (namespace equal to the current app package name), types or type-attributes of other modules $T_{A' \neq A}$ (namespace equal to some other app package), and system types or type-attributes T_S (system namespace). At installation time, the SEApp Policy Parser determines the origin of each type, with an explicit prohibition for policies to refer to types or type-attributes defined by other policy modules, while use of system types or type-attributes is subject to restrictions.

With regard to the `allow` statement, a dedicated analysis is performed by the SEApp Policy Parser. For each rule, the global origin of source and target types is determined. We refer to system origin *S*, when the type is directly or indirectly associated with a system type in the expansion of its definition, while to local origin *A* otherwise. Based on the origin of source and target of each rule, there are four cases. The case *AllowSS*, i.e., a permission with system origin both as source and target, is prohibited, as it represents a direct platform policy modification. The case *AllowAA* is always permitted, as it only defines access privileges internal to the app module. The cases *AllowAS* and *AllowSA* are more delicate.

An *AllowAS* originates when a local type needs to be granted a permission on a system type. A concrete example is shown in Section 3, where the *:media* process needs access to the `camera_service`. The case cannot be decided locally by the SEApp Policy Parser, therefore it is delegated to the SELinux decision engine during policy enforcement. This crucial postponed restriction depends on the constraint that all app types have to appear in a `typebounds` statement [32], which limits the bounded type to have at most the access privileges of the bounding type. As Android 11 assigns to generic third-party apps the `untrusted_app` domain, this is the candidate we use to bound the app types. If the *AllowAS* rule gives to the local type more privileges than those associated with `untrusted_app`, and at runtime these privileges are used, the SELinux decision engine identifies the policy violation and prohibits the action.

AllowSA rules are the key to regulate how system components access internal types. To be compliant with Android, the local types introduced by the app policy module must ensure interoperability with system services crucial to the app lifecycle. As an example *Zygote* [29], the native service which spawns and configures new app processes, can only execute processes labeled with the type-attribute `domain`, which is assigned by default to `untrusted_app`. However, giving app developers the freedom to directly define *AllowSA* rules would lead to two major issues: (i) the rules would depend on system policy internals, leading to a solution with limited abstraction and modularity; (ii) explicit *AllowSA* rules could lead to violations of the security assumptions of a system service, with the risk of introducing vulnerabilities (e.g., leading to a *confused deputy attack* [36]). For these reasons we prohibit their explicit use. To limit system types to only those already dealing with untrusted content and simplifying the policy, we rely on CIL macros, a set of function-like statements that, when invoked by the SEApp policy module, produce a predefined list of policy statements. This approach permits to retain control on the rules produced, ensuring no violation of the default system policy. Also, it makes the work of the developer easier, by abstracting away system policy internal details. To preserve the interoperability with system services, third-party app functionality has been broken down into the CIL macros listed in Table 2. This list has been identified looking at the internal structure of the `untrusted_app` domain. With this design philosophy, the developer can grant a basic set of permissions to a type (by calling one or more macros), and then add to it fine-grained authorizations with *AllowAS* rules.

Table 2: SEApp macros to grant permissions to local types

Macro	Usage
md_appdomain	to label app domains
md_netdomain	to access network
md_bluetoothdomain	to access bluetooth
md_untrusteddomain	to get full untrusted app permissions
mt_appdatafile	to label app files

With regard to the `typeattributeset` statement, the SEApp Policy Parser uses a verification strategy similar to the one used for allow rules. First, the global origin of the type-attribute and of the set expression of types and type-attributes is determined. All statements that directy or indirectly relate to system types are blocked. This avoids implicit permission propagation from system and local types.

Similarly, for the `typetransition` statement, the SEApp Policy Parser verifies the origin of the types involved, with a prohibition for all the statements that relate to system types, as they may lead to an escalation of privileges.

5 Policy configuration

In this section we explore the structure of application policy modules. Before describing the content of SEApp configuration files, we give a short description of how SEAndroid defines the security contexts of processes, files and system services. There are strong similarities between the structure of system and app policies. Indeed, we designed our solution as a natural extension of the approach used to protect the system. Also, our design maintains full backward compatibility. Developers who are not interested in taking advantage of MAC capabilities do not have to change their apps.

5.1 SEAndroid policy structure

Compared to a traditional Linux implementation, Android expands the set of configuration files where SELinux [18] security contexts are described, because a wider set of entities is supported. SEAndroid complements the common SELinux files (i.e., `file_contexts` and `genfs_contexts`) with 4 additional ones: `property_contexts`, `service_contexts`, `seapp_contexts` and `mac_permissions.xml`. Also, the implementation of the SELinux library (*libselinux*) [68] has been modified introducing new functions (to assign domains to app processes and types to their dedicated directory). We concisely describe the role of SEAndroid context files.

5.1.1 Processes

With reference to app processes, Android assigns the security context based on the class the app falls in. The specification of the classes and their security labels are defined in the `seapp_contexts` policy file. Most classes state two security contexts: one for the process (`domain` property) and the other one for the app dedicated directory (`type` property). A number of *input selectors* determine the association of an app with a class. Among these, `seinfo` filters on the tag associated with the X.509 certificate used by the developer to sign the app. The mapping between the certificate and the `seinfo` tag is achieved by the `mac_permissions.xml` configuration file. Since the enumeration of all third-party app certificates is not possible a priori, all third-party apps are labeled with the `untrusted_app` domain by default.

5.1.2 Files

SELinux splits the configuration of security contexts of files between `file_contexts` and `genfs_contexts`, with the former used with filesystems that support extended file attributes (e.g., `/data`), while the latter with the ones that do not (e.g., `/proc`). To apply `file_contexts` updates, two approaches are available: either rebuild the filesystem image, or run *restorecon* operation on the file or directory to be relabeled (this is the default method used by permissioned system processes). Conversely, to apply `genfs_contexts` changes, a reboot of the device or a sequence of filesystem *un-mount* and *mount* operations has to be performed.

5.1.3 Services

Unlike what happens for system processes, a system service requires the assignment of a security context to both its processes and its *Binder* [17], to be fully compliant with SEAndroid. The *Binder* is the lightweight inter-process communication primitive bridging access to a service. Its retrieval is enabled by the *servicemanager*, a process started during device boot-up to keep track of all the services available on the device. Based on the labels specified in the `service_contexts` file, it is then possible to control which processes can register (*add*) and lookup (*find*) a *Binder* reference for the service, and therefore connect to it. However, since *Binder* handles resemble tokens with almost unconstrained delegation, denying a process to get the *Binder* through the *servicemanager* does not prevent the process from obtaining it by other means (e.g., by abusing other processes that already hold it). Furthermore, preventing a process from obtaining a *Binder* reference prevents the process from using any functionality exposed by the service.

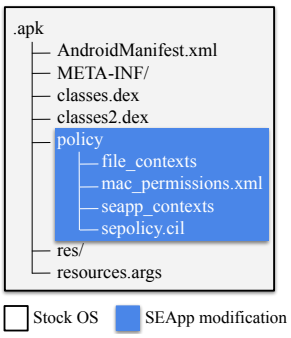


Figure 3: SEApp policy structure

5.2 SEApp policy structure

Developers interested in taking advantage of our approach to improve the security of their apps are required to load the policy into their Android Package (APK). A predefined directory, `policy`, at the root of the archive, is where the SEApp-aware package installer will be looking for the policy module (see Figure 3). Inside this directory, the installer looks for four files (which we refer to as *local*), that outline a policy structure similar to the one of the system. Specifically, the developer is able to operate at two different levels: (i) the actual definition of the app policy logic using the policy language described in Section 4 (in the local file `sepolicy.cil`), and (ii) the configuration of the *security context* for each process (in the local files `seapp_contexts` and `mac_permissions.xml`) and for each file directory (in the local file `file_contexts`).

5.2.1 Processes

SEApp permits to assign a SELinux domain to each process of the security enhanced app. To do this, the developer lists in the local `seapp_contexts` a set of entries that determine the security context to use for its processes. For each entry, we restrict the list of valid input selectors to `user`, `seinfo` and `name`: `user` is a selector based upon the type of UID; `seinfo` matches the app seinfo tag contained in the local `mac_permissions.xml` configuration file; `name` matches either a prefix or the whole process name. The conjunction of these selectors determines a class of processes, to which the

context specified by `domain` is assigned. To avoid privilege escalation, the only permitted domains are the ones the app defines within its policy module and `untrusted_app`. As a process may fall into multiple classes, the most selective one, with respect to the input selector, is chosen. An example of valid local `seapp_contexts` entries is shown in Listing 1, which shows the assignment of the *unclassified* and *secret* domains to the *:unclassified* and *:secret* processes, respectively.

In Android, developers have to focus on components rather than processes. Normally, all components of an application run in a single process. However, it is possible to change this default behavior setting the `android:process` attribute of the respective component inside the `AndroidManifest.xml`, thus declaring what is usually called a *remote* component. Furthermore, with the specification of an `android:process` consistent with the local `seapp_contexts` configuration, we support the assignment of distinct domains to app components. To execute the component, the developer is only required to create the proper *Intent* object [21], as she would have already done on stock Android for remote components. The assignment to the process of the correct domain is handled by the system. This design choice allows us to support Android activities, services, broadcast receivers and content providers, while avoiding changes to the *PackageParser* [62], as there are no modifications to the manifest schema.

5.2.2 Files

The developer states the SELinux security contexts of internal files in the local `file_contexts`. Each of its entries presents three syntactic elements, `pathname_regexp`, `file_type` and `security_context`: `pathname_regexp` defines the directory the entry is referred to (it can be a specific path or a regular expression); `file_type` describes the class of filesystem resource (i.e., directory, file, etc.); `security_context` is the security context used to label the resource. The admissible entries are those confined to the app dedicated directory and using types defined by the app policy module, with the exception of `app_data_file`. Due to the regexp support, a path may suit more entries, in which case the most specific one is used. Examples of valid local `file_contexts` entries are shown in Listing 2: the first line describes the default label for app files, second and third line respectively specify the label for files in directories `dir/unclassified` and `dir/secret`.

In SELinux, the security context of a file is inherited from the parent folder, even though `file_contexts` might state otherwise. Since, for our approach, it is essential that files are labeled as expected by the developer, we decided to enforce file relabeling at creation. Therefore, a new native service has been added to the system (see Section 6.2). We then offer to the developer an alternative implementation of class `java.io.File`, named `android.os.File`, which sets file and directory context upon its creation, transparently handling the call to our service.

5.2.3 System services

To support any third-party app, the `untrusted_app` domain grants to a process the permissions to access all system services an app could require in the `AndroidManifest.xml`. As an example, in Android 11, the `untrusted_app_all.te` platform policy file [28] permits to a process labeled with `untrusted_app` to access `audioserver`, `camera`, `location`, `mediaserver`, `nfc` services and many more.

To prevent certain components of the app from holding the privilege to bind to unnecessary system services, the developer defines a domain with a subset of the `untrusted_app` privileges (in the local `sepolicy.cil` file), and then she ensures the components are executed in the process labeled with it. Listing 3 shows an example in which the `cameraserver` service is made accessible to the *secret* process.

```
1 user=_app seinfo=cert_id domain=package_name.
   unclassified name=package.name:unclassified
2 user=_app seinfo=cert_id domain=package_name.
   secret name=package.name:secret
```

Listing 1: `seapp_contexts` example

```
1 .*                u:object_r:app_data_file:s0
2 dir/unclassified  u:object_r:package_name.
   unclassified_file:s0
3 dir/secret        u:object_r:package_name.
   secret_file:s0
```

Listing 2: `file_contexts` example

```
1 (block package_name
2   (type secret)
3   (call md_appdomain (secret))
4   (typebounds untrusted_app secret)
5   (allow secret cameraserver_service (
     service_manager (find)))...)
```

Listing 3: Granting `cameraserver` access to secret domain

6 Implementation

In this section we describe the main changes introduced in Android by SEApp. We first analyze the modifications required to manage policy modules, both during device boot and at app installation. We then describe how the runtime support was realized.

6.1 Policy compilation

6.1.1 Boot procedure

Since the introduction of Project Treble [10], policy files are split among multiple partitions, one for each device maintainer (i.e., platform, SoC vendor, ODM, and OEM). This feature facilitates updates to new versions of Android, separating the Android OS Framework from the device-specific low-level software written by the chip manufacturers. Yet, each time a partition policy (i.e., a segment) changes, an on-device compilation is required.

The *init* process divides its operations in three stages [19]: (i) *first stage* (early mount), (ii) SELinux setup, and (iii) *second stage* (init.rc). The first stage mounts the essential partitions (i.e., `/dev`, `/proc`, `/sys` and `/sys/fs/selinux`), alongside some other partitions specified as early mounted (since Android 10 using an `fstab` file in the first stage ramdisk, in Android 9 and lower adding fstab entries using device tree overlays). Once the required partitions are mounted, *init* enters the SELinux setup. As the name suggests, this is the stage where *init* loads the SELinux policy. As the `/data` partition, where policy modules are stored, is not yet mounted, it is not yet possible to integrate them with the policy of the system. Then, as last operation of the SELinux setup stage, *init* re-executes itself to transition from the initial `kernel` domain to the `init` domain, entering the second stage. As the second stage starts, *init* parses the `init.rc` files and performs the builtin functions listed there, among them mounting the `/data` partition. Now, the policy modules are available, and we can produce with *secilc* [26] (the SELinux CIL compiler) the binary policy consisting of the integration among the system policy, the SEApp macros and the app policy modules. To trigger the build and reload of the policy, we implemented a new builtin function, and modified the `init.rc` to call this function right after `/data` is mounted. The policy is considered immediately after the `/data` partition is available and this ensures that the policy modules are loaded far before an application starts, making the policy not bypassable.

Even though most Android devices supporting Android 10 were released with Treble support and, therefore, execute their SELinux setup stage on the `sepolicy.cil` fragments scattered among multiple partitions, *init* still supports the use of a legacy monolithic binary policy. For compatibility towards devices using a monolithic binary policy, additional changes are required, as SEApp needs the system policy written in CIL to be compiled alongside with app modules. To this end, we modified the Android build process to push the `sepolicy.cil` files onto the device even for non-Treble devices. New entries in the device tree were added to make the policy segments available during *init* SELinux setup stage [22].

As previously mentioned, we decided to store the policy modules in the `/data` partition; even if this choice required us to adapt the boot procedure of the device, it smoothly integrates SEApp with the current Android design. In fact, the `/data` partition is one of the few writable partitions, it is dedicated to hold the APK the user installs, as well as their dedicated data directories and, therefore, it represents the best option to contain also the app policy modules. Moreover, whenever a user performs a factory reset, Android automatically wipes the `/data` partition, removing the customization the user made to the device configuration, including the apps. By placing the app policy modules and the apps into the same partition, a factory reset removes the policy modules as well.

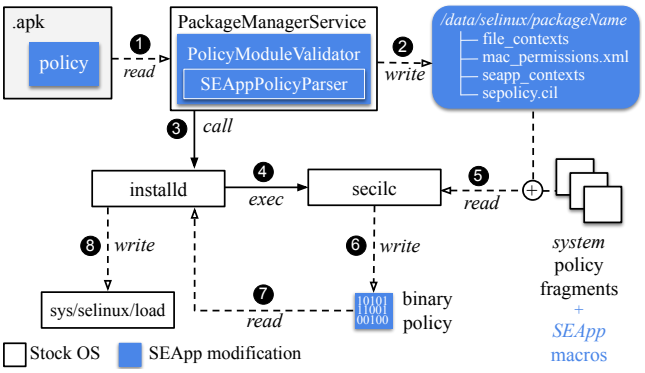


Figure 4: Installation process

6.1.2 App installation

As introduced in Section 5.2, the developer willing to define its own policy module is expected to load it in the app package. At app installation, the *PackageManagerService* [23] inspects the APK to identify whether or not the current installation involves a policy module, by looking for the `policy` directory at the root of the archive. When the app has a policy module attached to it (see Figure 4), the *PackageManagerService* extracts it (❶) and uses our *PolicyModuleValidator* to verify the respect of all the constraints on `sepolicy.cil` (through the *SEAppPolicyParser*, Section 4) and on the configuration files (Section 5). In case of a violation of the constraints, the app installation stops. Otherwise, the policy module is stored within `/data/selinux`, in a dedicated directory identified by the package name (❷). Then, the *PackageManagerService* invokes *installd* [20] through the *Installer* to trigger the policy compilation with an exec call to the *secilc* program (❸, ❹). *Secilc* reads the system `sepolicy.cil` fragments, the SEApp macros and the `sepolicy.cil` fragments of the app policy modules in the `/data/selinux` directory (❺), and builds the binary policy (❻). When the *secilc* execution returns and no compilation errors have been raised, the binary policy is then read by *installd* (❼) and loaded with *selinux_android_load_policy*, which writes the `sys/selinux/load` file (❽).

To load the policy files after *init*, the implementation of SELinux in Android has been slightly modified. In particular, we modified the policy loading function within *libselinux* (function *selinux_android_load_policy*), and changed the system policy to allow *installd* to load the app policy module.

As for the policy configuration files, some changes were introduced to properly load the application `file_contexts`, `seapp_contexts` and `mac_permissions.xml`. *SELinuxMMAC* [27], i.e., the class responsible for loading the appropriate `mac_permissions.xml` file and assigning `seinfo` values to apks, was modified to load the new `mac_permissions.xml` specified within the app policy module. The loading of `file_contexts` and `seapp_contexts`

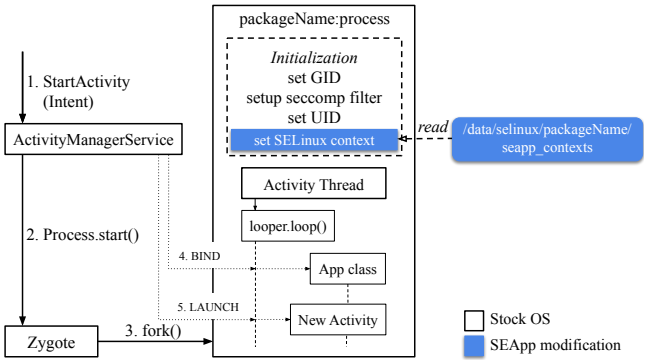


Figure 5: Application launch

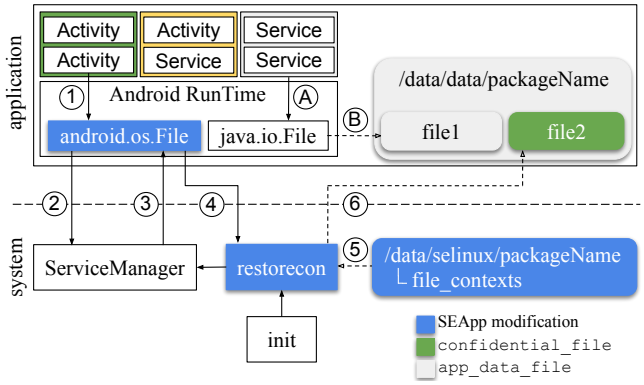


Figure 6: File relabeling

was configured to treat system and app configuration files apart. So, SEApp-enhanced applications will load exclusively their configuration files, whereas the loading of system's and other apps' configuration files is not needed since their use is prohibited. System services and daemons, instead, load the base system configurations once, and then load the app policy module specific configuration files as they are needed. An example of this are *Zygote* and *restorecon* services, which need to retrieve at runtime `seapp_contexts` and `file_contexts`, respectively (see Section 6.2).

Our implementation also supports the uninstallation of SEApp apps. The regular uninstallation process is extended with a step where the global policy is recompiled, in order to remove the impact of old modules on the overall binary policy. With reference to application updates, the native *installd* runs with the necessary permission to remove and apply new file types based on the content of the `file_contexts`.

6.2 Runtime support

In addition to the steps described above, other aspects have to be considered in order to extend SELinux support at the application layer.

6.2.1 Processes

Android application design is based on components. Each of them lives inside a process, and can be seen as an entry point through which the system or the user can enter the app.

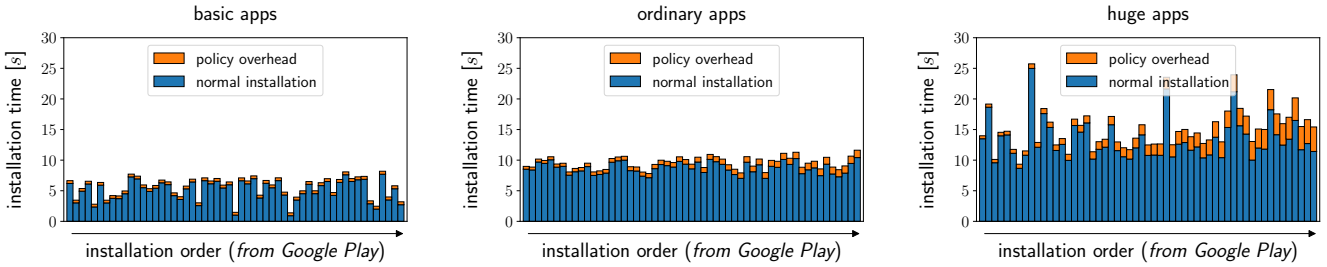
To activate a component, an asynchronous message called intent, containing both the reference to the target component and parameters needed for its execution, has to be created. The intent is then routed by the system to the *ActivityManagerService* [12] via *Binder IPC*. Before delivering the intent request to the target component, the *ActivityManagerService* checks if the process in which the target component should be executed is already running; if not, the native service called *Zygote* [29] is executed. Its role is to spawn and correctly setup the new application process. To achieve this, it first replicates itself by performing a fork, then, using the input provided by the *ActivityManagerService* (namely, package name, `seinfo`, `android:process`, etc.), it starts configuring the process GID, the *seccomp* filter, the UID and finally the SELinux security context. We adapted the final configuration step, forcing *Zygote* to set the security context based on the `seapp_contexts` located at `/data/selinux/packageName` (i.e., the one provided by the developer for her app). Process name is used to assign the proper context to the process when it starts, before the logic of the process kicks in. In case the developer did not specify a domain, then *Zygote* uses the system `seapp_contexts` as fallback. After the correct labeling, the *ActivityManagerService* finishes the configuration by binding the application class, launching the component, and finally delivering the intent message. Figure 5 details the process.

This implementation design offers several benefits, including backward compatibility, support for all components, and ease of use. Indeed, a developer who wants to use our solution only has to configure some files; changes in the application code are reduced to a minimum, thus facilitating the introduction of SELinux in already existing apps.

In our study we have also explored other design alternatives, in which the developer could explicitly state a domain transition in the code, wherever she needs it. Although this category of solutions would give the developers more control over domain transitions, it also has some drawbacks. First, the developer would be expected to enforce the isolation among source and target domains managing the multi-threaded scenario, and second, this design implies granting too many permissions to the app (e.g., `dyntransition`, `setcurrent` and read/write access to `selinuxfs`). Moreover, such solution would introduce a new Android API, that would be quite delicate and, if not used correctly, it might be difficult to control.

6.2.2 Files

Android applications aiming to create a file can use the `java.io.File` abstraction. Each file creation request that is generated is captured by the Android Runtime (ART) [15], and then converted into the appropriate syscall. The result



(a) (b) (c)

Figure 7: Installation time overhead for apps with different complexity

is the creation of the target file, to which a security context inherited from the parent directory is assigned (see flow Ⓐ, Ⓑ of Figure 6). Since Android 9, the separation between files of different apps is enforced at MAC level (a unique context based on UID and SELinux category is assigned); however, all the files stored in the same app folder are labeled with the `app_data_file` type.

To make the most out of SELinux, SEApp complements Android with the implementation of a new service, which we called *restorecon* (to recall the SELinux *restorecon.c* tool). The *restorecon* service is spawned by *init* at boot, and works in its own SELinux domain. Its role is to create and label files as specified by the developer in the local `file_contexts`. To ease development, we implemented the new `android.os.File` abstraction, which exposes an interface equal to that of `java.io.File`, and transparently handles the call to our service. Figure 6 details the new control flow. A component running in a SEApp-enhanced process (highlighted in green in Figure 6) invokes `android.os.file`, and triggers a new file creation request (①). The new API first interacts with the *ServiceManager* (②) to get a handle of the *restorecon* service (③), then it interacts with the service using the AIDL [5] interface we defined for it, informing the *restorecon* of the target path (④). The *restorecon* service verifies whether the caller is the legitimate owner of the path, it reads the `file_contexts` file located at `/data/selinux/packageName` (⑤), and finally it creates the target file enforcing the correct labeling (⑥).

We also investigated three other implementation approaches: (i) change of the default security context inheritance behavior for the *ext4* filesystem, (ii) execution of the SELinux *restorecon* operation by the app, once the file is successfully created, and (iii) use of *restorecond* [25]. The first option would change the default behavior system-wide. As it might cause compatibility issues, we decided not to choose it. The second option is not ideal from a security standpoint, as it requires to grant the application too many permissions (e.g., `relabelfrom`, `relabelto`, as well as read/write access to `selinuxfs` to check the validity of the SELinux context). The third option refers to the use of *restorecond*, a system daemon that watches (inodes of) a configurable list of files and checks that they are labeled as stated in the system `file_contexts`. Although it may realize the control, *restorecond* was meant for a few system files, therefore its performance would hardly scale, especially considering that SEApp needs to manage all files created by SEApp-aware apps. Another major issue is that this approach is exposed to race conditions, because there is a delay between file creation and its relabeling.

7 Performance

We now present a performance evaluation of SEApp. The experiments have been conducted on both Android 9 and 10, each with Linux kernel v4.9. However, all the measurements shown refer to Android 10 (release android-10.0.0_r41). The device used to run the tests is a Google Pixel 3 (*blueline*), in which the four *gold* cores frequency was set to 2.8 GHz, while the four *silver* ones were disabled. The change in CPU configuration has been performed to reduce the variability of measures. The confidence intervals provided have an associated confidence level of 99%.

7.1 App installation

The introduction of dedicated app policies implies further steps to be executed at app installation time, as each SEApp module has to be validated, compiled, and loaded. To evaluate the impact on performance, we wrote dedicated tests to stress the installation procedure with multiple application samples.

To build representative samples of a typical consumer scenario, we first downloaded the 150 most popular free apps from Google Play (retrieved in October 2020) [52]. The apps were subsequently divided into three buckets: *basic*, *ordinary* and *huge* apps, according to the weighted normalized average of the .apk size, the number of Android activities and the number of services. Based on the bucket, each app was equipped with one of the following policy configurations: (i) *basic*, 1 domain and 1 type per policy module, (ii) *ordinary*, 10 domains and 25 types, and (iii) *huge*, 20 domains and 100 types. The rationale is that larger apps can gain considerable

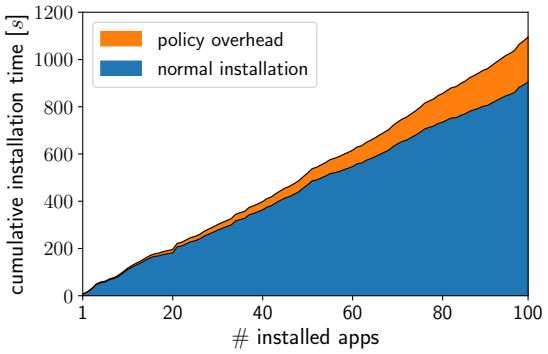


Figure 8: Cumulative install time overhead when installing the top 100 free apps on Google Play Store with our policies

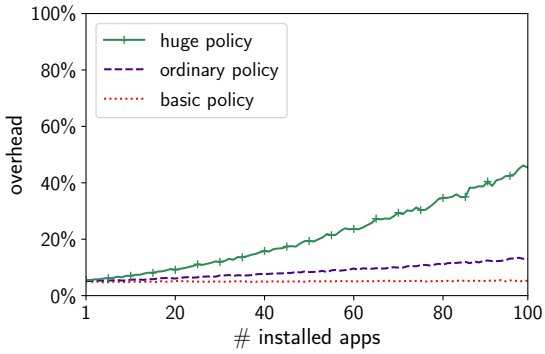


Figure 9: Install time overhead for the three policy sizes

benefit from the use of a large policy. The *basic* configuration mimics how third-party apps are currently handled, but with some key improvements, as it permits to define the subset of services the domain can use, and it permits to enforce app isolation, not only based on MAC category, but also through the specification of its own type. The *ordinary* and *huge* policy configurations are meant to take full advantage of intra-app isolation and flexibility via the definition of multiple domains. Each test was repeated five times, measuring the time each package took to install. The measurements were done with the *nix *date* utility.

Test I. To measure the overhead caused by the presence of the policy module, we performed on device installation of each of the previously described app buckets (*basic*, *ordinary* and *huge*) via Android Debug Bridge (adb) [13].

The results of Test I are illustrated in Figure 7. In detail, it shows in blue (i.e., the lower part of the bar) the time required by the system to install the current package without the dedicated policy module, while in orange (i.e., the top of the bar) the overhead caused by the presence of the policy module. The data report that a limited overhead is associated with apps with *huge* policies, at most $3.59 \pm 0.04s$, while *basic* and *ordinary* policy configurations exhibit a negligible slowdown, never exceeding $1.22 \pm 0.02s$.

Test II. To evaluate the overall impact of SEApp in a typical consumer scenario, we performed a test evaluating cumulative installations. At first, we repeated the installation of the top 100 apps on Google Play Store with the same policy configuration as in Test I (see Figure 8). In this case, we measured an overhead of $20.98 \pm 1,31\%$ on total installation time.

As explained in Section 6, each time a new application is installed, all policy fragments stored in the device have to be recompiled to produce the new binary policy. The installation time overhead then grows with the increase in the number of installed policy modules. To further analyze this aspect, we repeated the installation of the top 100 free apps adding to all the packages in three separate experiments the same *basic*, *ordinary*, and *huge* policy configurations. The experimental results illustrated in Figure 9, show that only the use of *huge* policy modules introduces a non-negligible overhead ($45.35 \pm 2.44\%$ on total installation time). However, this policy configuration simulates an edge case, as we do not expect to find 100 of them in a real scenario. To give a comparison, the *huge* policy declares 100 types; `public/file.te`, i.e., the file used to define all the file types of the system, declares 314 types in Android 10.

In Table 3 we report the sizes of the overall policies for the three scenarios considered in this experiment. We report the number of MAC types, the number of produced AV rules, and the overall size in KBytes of the binary policy.

Table 3: Policy size

policy	#types	#avrules	KB
system	1536	29228	596
system + 100 basic	1836	47028	867
system + 100 ordinary	6036	213228	3512
system + 100 huge	15536	417228	7064

7.2 Runtime performance

We now evaluate the runtime overhead for an app taking full advantage of SEApp. We focus on the creation of processes and files, as they are the entities directly affected by the changes made in the implementation. The data shown refer to the creation time of each resource. The measurements have been acquired via *System.nanoTime* and have been repeated 100 times for each test. Also, all outliers diverging more than 3 standard deviations from the mean have been suppressed.

7.2.1 Processes

As discussed in Section 6, in SEApp the creation of a process is originated from the request of execution of an Android component. Thus, the slowdown occurs between the request for the component and the execution of the method *onCreate*, which is the time interval subject to measurement. Our evaluation is limited to activities and services, as these are the com-

Table 4: Cold and warm start performance for activities and services

Component	Cold start (ms)				Warm start (ms)			
	Stock OS		*SEApp*		*Stock OS*		*SEApp*	
	μ	σ	μ	σ	μ	σ	μ	σ
LocalActivity	39.102	1.094	38.689	0.980	21.052	6.046	18.685	5.001
RemoteActivity	123.468	3.176	124.649	3.526	15.722	2.682	15.933	3.256
SEApp Activity	-	-	127.356	3.542	-	-	15.188	2.394
LocalService	19.164	1.444	18.835	1.392	1.399	0.208	1.328	0.208
RemoteService	105.467	2.800	106.935	2.565	2.617	0.879	2.676	0.593
IsolatedService	103.923	2.425	104.260	3.727	-	-	-	-
SEApp Service	-	-	106.925	3.774	-	-	2.528	0.675

Table 5: File creation performance

File creation		
Test	μ (µs)	σ (µs)
Stock OS	57.077	5.174
SEApp	60.696	6.782
SEApp + *restorecon*	431.472	109.494

ponents most used by developers. Our analysis showed identical behavior for broadcast receivers and content providers, the other two components supporting the `android:process` attribute in the manifest.

Separate test cases have been identified based on the type of process that supports the component. We refer to *Local*, *Remote*, *Isolated* or *SEApp* components when we run components respectively in the current process, in another process, in another process with the `isolated_app` domain (using the *isolatedprocess* we described in Section 3.1.3), or in a package specific domain (declared in the app policy module). Furthermore, we cover *cold* and *warm* start scenarios. The *cold* start corresponds to the first time the application brings up the component, and the *warm* start to the subsequent times the app reuses a previously instantiated one.

The results shown in Table 4 demonstrate that the performance of a stock version of the OS and SEApp are equivalent. Also, we observe that apps willing to benefit of the intra-app isolation feature get from the use of SEApp the same performance they would get from the use of remote components. Our approach also proves to outperform the *IsolatedService*, as the *isolatedprocess* option forces the creation of a new process every time an *IsolatedService* that was previously *unBind*-ed is activated. This introduces a slowdown of $102 \pm 1ms$ compared to the *SEAppService* warm start, which instead benefits from the system caching mechanism.

7.2.2 Files

Alongside the usual creation method, SEApp introduces in Android the possibility of creating files with a security domain defined by the app dedicated `file_contexts`. Table 5 shows the time required to create a file, for each of the methods discussed. We observe no overhead on direct file creation, but the overall execution time becomes larger due to the invocation, as described in Section 6.2, of the *restorecon* service, which requires approximately $374 \pm 30\mu s$. This overhead only occurs at file creation and every subsequent operation on the file does not exhibit any performance degradation.

8 Related work

In traditional desktop operating systems significant effort has been spent in retrofitting legacy code for authorization policy enforcement leveraging MAC. An approach is to place reference monitor calls to mediate sensitive access locations through the use of static and dynamic analysis [49, 59]. An evolution of this solution is the multi-layer reference monitor [54], in which the MAC policy is enforced at different levels (e.g., application, OS, Virtual Machine Manager). Another approach is to identify integrity-violating permissions through the use of information-flow analysis [64].

Android's open source nature and popularity made it the target of careful security investigations (e.g., [1, 42, 43, 47]) and several proposals aiming at strengthen its security properties. In the following we discuss the ones that try to address app isolation and modularity, underlining the key differences with our methodology.

Our approach presents similarities with *Secure Application INTeraction (Saint)* proposed by Ongtang et al. in [60], in which the authors also try to address the issue of allowing developers to define policies that can be verified at both installation time and runtime, to better specify the permissions for each component of their app. However, since the paper has been published in 2010, *Saint* could not leverage SEAndroid [65], which was introduced later, thus the authors had to define their own Android security middleware, which would not fit into the current Android architecture [58].

FlaskDroid [37] defines a versatile middleware and kernel layer policy language. It is based on Userspace Object Managers (USOMs), which control access to services, intents and data stored in Content Providers. However, FlaskDroid does not focus on intra-app compartmentalization, a central aspect in our proposal.

ASM [53] and *ASF* [34] promote the need for a programmable interface that could serve as a flexible ecosystem for different security solutions. The generality of these solutions, however, requires to introduce several changes to the current Android security model.

AppPolicyModules [31] is another proposal that allows app developers to create dedicated policy modules. The authors

focus more on how apps could use SEAndroid to better protect their resources from the system and from other apps, paying limited attention to internal compartmentalization.

DroidCap [39] is a recent contribution proposed by Dawoud and Bugiel, in which the authors propose to replace Android's UID-based ambient authority (DAC) with per-process Binder object capabilities. The proposal is interesting as it permits to achieve security compartmentalization between different app components. To introduce capability-based access control on files, *DroidCap* had to integrate *Capsicum for Linux* [50] in Android. Overall, DroidCap is a nicely engineered solution, which shares similar objectives with ours, and the two could work in parallel as they do not interfere with each other. However, as our proposal relies on SELinux and SEAndroid, which are already part of the Android security framework, our architecture appears to be more aligned with the natural evolution of the Android ecosystem.

Boxify [35] is a virtualization environment for Android apps, which could be used to achieve a higher level of privacy and better control over app permissions. The authors also describe how their solution could be used to compartmentalize Ads libraries to reduce the risk of sensible information leakage. Yet, since the virtualization environment acts as a mediator between the applications and the system, it extends the set of trusted components the app has to rely on.

AFrame [71] and *CompARTist* [55] propose to compartmentalize third-party libs from their host app using a separate process with a dedicated UID. In *AFrame* the Android Manifest is modified with the introduction of library ad-hoc permissions, while *CompARTist* uses compile time app rewriting. Both proposals do not extend the protection at the MAC level.

To summarize, the main differences that characterize our proposal are: (i) we propose a natural extension of the role of SELinux to apps leveraging what is already used to protect the system itself, thus minimizing the impact on it, and (ii) we empower the developers while limiting the amount of changes an application must undergo in order to take advantage of our solution.

9 Conclusions

In this paper we proposed an extension to the current MAC solution (SELinux) already available in Android. Developers can use SELinux to define domains that are internal to their apps, in such a way that it is possible to leverage the modules that are already providing protection to the system. By mapping SELinux domains to activities and services, developers can limit the impact that a vulnerability has on the app processes and files. We described in the paper the changes that we introduced into Android, and our experimental evaluation shows that the overhead introduced by our proposal is compatible with the additional security guarantees.

Acknowledgments

We thank our shepherd Sven Bugiel and the anonymous reviewers for their valuable comments and feedback. This work was supported in part by the European Commission under grant agreement No 825333 (MOSAICrOWN), and by the 2015 Google Faculty Research Award Program.

Availability

The implementation source and artifacts produced for the evaluation of our proposals are freely available at this URL: https://github.com/matthewrossi/seapp

References

[1] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. SoK: Lessons learned from Android security research for appified software platforms. In *IEEE S&P*, 2016.

[2] Android. Google Play Protect. https://www.android.com/play-protect/, 2021.

[3] Android Developers. adb install. https://developer.android.com/studio/command-line/adb#move, 2021.

[4] Android Developers. Android App Bundles. https://developer.android.com/platform/technology/app-bundle, 2021.

[5] Android Developers. Android Interface Definition Language. https://developer.android.com/guide/components/aidl, 2021.

[6] Android Developers. android:isolatedProcess. https://developer.android.com/guide/topics/manifest/service-element#isolated, 2021.

[7] Android Developers. Bound services overview. https://developer.android.com/guide/components/bound-services#Creating, 2021.

[8] Android Developers. isolated_app.te. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/isolated_app.te, 2021.

[9] Android Open Source Project. Enable per-user isolation for normal apps. https://android.googlesource.com/platform/external/sepolicy/+/a833763ba04147e840fd054b613f759395bada35, 2014.

[10] Android Open Source Project. SELinux for Android 8.0. https://source.android.com/security/selinux/images/SELinux_Treble.pdf, 2017.

[11] Android Open Source Project. Android 9 release notes. https://source.android.com/setup/start/p-release-notes#per-app_selinux_sandbox, 2018.

[12] Android Open Source Project. ActivityManagerService. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/am/ActivityManagerService.java, 2021.

[13] Android Open Source Project. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb, 2021.

[14] Android Open Source Project. Android Permissions. https://developer.android.com/guide/topics/permissions/overview, 2021.

[15] Android Open Source Project. Android Runtime. https://developer.android.com/guide/platform#art, 2021.

[16] Android Open Source Project. App manifest overview. https://developer.android.com/guide/topics/manifest/manifest-intro, 2021.

[17] Android Open Source Project. Binder. https://developer.android.com/reference/android/os/Binder, 2021.

[18] Android Open Source Project. Implementing SELinux. https://source.android.com/security/selinux/implement, 2021.

[19] Android Open Source Project. init. https://android.googlesource.com/platform/system/core/+/refs/heads/master/init/main.cpp, 2021.

[20] Android Open Source Project. installd. https://android.googlesource.com/platform/frameworks/native/+/refs/heads/master/cmds/installd/, 2021.

[21] Android Open Source Project. Intent and intent filters. https://developer.android.com/guide/components/intents-filters, 2021.

[22] Android Open Source Project. Mounting partitions early. https://source.android.com/devices/architecture/kernel/mounting-partitions-early, 2021.

[23] Android Open Source Project. PackageManagerService. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/PackageManagerService.java, 2021.

[24] Android Open Source Project. Policy compatibility. https://source.android.com/security/selinux/compatibility, 2021.

[25] Android Open Source Project. restorecond service. https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/restorecond/restorecond.service, 2021.

[26] Android Open Source Project. secilc. https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/secilc/, 2021.

[27] Android Open Source Project. SELinuxMMAC. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/SELinuxMMAC.java, 2021.

[28] Android Open Source Project. untrusted_app_all.te. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/untrusted_app_all.te, 2021.

[29] Android Open Source Project. Zygote. https://android.googlesource.com/platform/frameworks/base.git/+/master/core/java/com/android/internal/os/Zygote.java, 2021.

[30] Ars Technica. The Android 11 interview. https://arstechnica.com/gadgets/2020/09/the-android-11-interview-googlers-answer-our-burning-questions/, 2020.

[31] E. Bacis, S. Mutti, and S. Paraboschi. AppPolicyModules: Mandatory access control for third-party apps. In *ASIACCS*, 2015.

[32] E. Bacis, S. Mutti, and S. Paraboschi. Policy specialization to support domain isolation. In *SafeConfig*, 2015.

[33] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in Android and its security applications. In *CCS*, 2016.

[34] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on Android. In *ACSAC*, 2014.

[35] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. Von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock Android. In *USENIX Security*, 2015.

[36] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[37] S. Bugiel, S. Heuser, and A.R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security*, 2013.

[38] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang. Analysis of SEAndroid policies: Combining MAC and DAC in Android. In *ACSAC*, 2017.

[39] A. Dawoud and S. Bugiel. DroidCap: OS support for capability-based permissions in Android. In *NDSS*, 2019.

[40] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C.A. Gunter. Free for all! Assessing user data exposure to advertising libraries on Android. In *NDSS*, 2016.

[41] M. Diamantaris, E.P. Papadopoulos, E. Markatos, S. Ioannidis, and J. Polakis. REAPER: Real-time app analysis for augmenting the Android permission system. In *CODASPY*, 2019.

[42] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security*, 2011.

[43] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE S&P Magazine*, 2009.

[44] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions demystified. In *CCS*, 2011.

[45] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[46] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? The impact of copy paste on Android application security. In *IEEE S&P*, 2017.

[47] Y. Fratantonio, A. Bianchi, W. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna. On the security and engineering implications of finer-grained access controls for Android developers and users. In *DIMVA*, 2015.

[48] Free Software Foundation. GNU M4. https://www.gnu.org/savannah-checkouts/gnu/m4/manual/m4-1.4.18/index.html, 2016.

[49] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE S&P*, 2006.

[50] Google. Capsicum object-capabilities on Linux. https://github.com/google/capsicum-linux, 2017.

[51] Google Play Protect. Android app vulnerability classes: A whirlwind overview of common security and privacy problems in Android apps. https://static.googleusercontent.com/media/www.google.com/en//about/appsecurity/play-rewards/Android_app_vulnerability_classes.pdf, 2021.

[52] Google Play Store. Android top apps. https://play.google.com/store/apps/top, 2021.

[53] S. Heuser, A. Nadkarni, W. Enck, and A.R. Sadeghi. ASM: A programmable interface for extending Android security. In *USENIX Security*, 2014.

[54] R. Hicks, S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, and T. Jaeger. An architecture for enforcing end-to-end access control over web applications. In *SACMAT*, 2010.

[55] J. Huang, O. Schranz, S. Bugiel, and M. Backes. The ART of app compartmentalization: Compiler-based library privilege separation on stock Android. In *CCS*, 2017.

[56] J. Vander Stoep. ioctl command whitelisting in SELinux. http://kernsec.org/files/lss2015/vanderstoep.pdf, 2015.

[57] K. MacMillan, C. Case, J. Brindle, and C. Sellers. SELinux Common Intermediate Language motivation and design. https://github.com/SELinuxProject/cil/wiki, 2020.

[58] R. Mayrhofer, J. Vander Stoep, C. Brubaker, and N. Kralevich. The Android platform security model. *arXiv*, 2019.

[59] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging "choice" to automate authorization hook placement. In *CCS*, 2012.

[60] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *ACSAC*, 2009.

[61] P. Pearce, A.P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *ASIACCS*, 2012.

[62] Android Open Source Project. PackageParser. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/content/pm/PackageParser.java, 2021.

[63] R. Sandhu and P. Samarati. Authentication, access control, and audit. *CSUR*, 1996.

[64] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *NDSS*, 2006.

[65] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, 2013.

[66] Statista. Most popular installed ad network software development kits (SDKs) across Android apps worldwide as of September 2020. https://www.statista.com/statistics/1035623/leading-mobile-app-ad-network-sdks-android/, 2020.

[67] The SELinux Project. Type Enforcement. https://selinuxproject.org/page/NB_TE, 2015.

[68] The SELinux Project. libselinux. https://github.com/SELinuxProject/selinux/tree/master/libselinux, 2021.

[69] Unity. Unity Ads. https://unity.com/solutions/unity-ads, 2021.

[70] C. Wright, C. Cowan, J. Morris, James, S. Smalley, and G. Kroah-Hartman. Linux Security Module framework. In *Ottawa Linux Symposium*, 2002.

[71] Z. Xiao, A. Amit, and D. Wenliang. AFrame: Isolating advertisements from mobile applications in Android. In *ACSAC*, 2013.

[72] Zerodium. Zerodium - The leading exploit acquisition platform. https://zerodium.com, 2021.

A Application of SEApp

In this Section we give a technical demonstration of the security measures introduced by SEApp. The description is based on the showcase app presented in Section 3. We show that: (1) the showcase app can operate without a policy module; in this mode, its vulnerabilities can be exploited; (2) the showcase app can also operate with the policy module listed in Appendix A.4 and use the services offered by SEApp; in this mode, the internal vulnerabilities are no longer exploitable.

The showcase app has a minimal structure. Its entry point is the *MainActivity*, which is associated with the *core_logic* process. From the *MainActivity* it is possible to send a *startActivity* intent to one among *UseCase1Activity*, *UseCase2Activity* and *UseCase3Activity*; the entry points of use cases 1, 2 and 3, respectively. For each entry point *Zygote* starts a dedicated process and, according to the content of the `seapp_contexts` (in Listing 4), assigns its specific domain (`user_logic_d` to UC#1, `ads_d` to UC#2, `media_d` to UC#3). A dedicated description of each use case follows.

A.1 Use case 1

In this use case we demonstrate how an app could benefit from the fine-granularity access to files. In particular, we

show how the *UseCase1Activity*, suffering of a path traversal vulnerability, cannot be exploited when the app is associated with a properly configured policy module. According to the Google Play Protect report on common application vulnerabilities [51], unsanitized path names that lead to path traversal are a primary source of problems in applications.

UseCase1Activity is quite simple: it displays the content of a file given its relative path through an intent. While this may be fine when the intent comes from trusted components, the activity supports also implicit intents coming from untrusted sources. This makes the vulnerability easily exploitable by an attacker targeting the confidential files written by the *core_logic* components.

In our setup phase, we leverage *MainActivity* to create an internal directory structure by using the `android.os.File` abstraction, which sets file and directory context upon its creation (see Section 6.2.2). Two directories are created: `user/` and `confidential/`; inside both folders a file `data` is saved.

To test this use case, we first start *UseCase1Activity*, then we send an intent to "confuse" *UseCase1Activity* into showing us the content of `confidential/data`. This can be done via ADB with the command:

```
adb shell am start
-n com.example.showcaseapp/.UseCase1Activity
-a "com.example.showcaseapp.intent.action.SHOW"
--es "com.example.showcaseapp.intent.extra.PATH"
 "../confidential/data"
```

When the policy module is missing, all internal files are flagged with `app_data_file` and every app component executes within the `untrusted_app` domain, which holds read access to `app_data_file`. As a consequence the vulnerability is successfully exploited and *UseCase1Activity* shows the content of the `confidential/data` file.

Instead, when the policy module is enforced by SEApp, the file `confidential/data` is flagged with `confidential_t`, as indicated in line 2 in `file_contexts` (see Listing 5). Since no permission is granted on `confidential_t` in the `sepolicy.cil` to `user_logic_d`, any access to the file `confidential/data` by *UseCase1Activity* is blocked by SELinux. The following denial is written to the system log: *denied* `search` *to* `user_logic_d` *domain on* `confidential_t` *type*. The `confidential` directory cannot then be accessed despite the exploitation of the path traversal vulnerability.

A.2 Use case 2

In this use case we show how to confine an Ad library into an ad-hoc process, with guarantees that it cannot abuse the access privileges granted to the whole application sandbox by the user. To do that, we deliberately inject, in the same process the library is executed, a malicious component (which is directly invoked by the library) that tries to capture the location when the permission *ACCESS_FINE_LOCATION* is granted to the app. The Ad library used is Unity Ads [69], which according to [66] in 2020 was used by 11% of apps that show ads.

In this case the library is invoked by *UseCase2Activity*, and according to line 3 of the `seapp_contexts`, both the activity and the components created by the library are executed by *Zygote* in a process labeled with `ads_d`. To interact with the Ad library, *UseCase2Activity* instances a `UnityAdsListener`. After the Ad initialization (including the registration of the listener) and displaying the Ad to the user, the Ad framework invokes the listener callback method `onUnityAdsFinish`, which executes the malicious routine `captureLocation`. The routine probes the app permissions; if *ACCESS_FINE_LOCATION* was granted to the app, the malicious component retrieves through the *servicemanager* a handle to the `LocationManager`, and registers to it an asynchronous listener that captures GPS location.

We show that when the policy module is enforced by SEApp, the malicious component cannot access the GPS coordinates. This is because the component is executed in the same process of the library, which is labeled with `ads_d`. If we look at the `sepolicy.cil` (lines 43-50), `ads_d` is not granted access to the SELinux type `location_service`, so the malicious routine cannot retrieve and therefore connect to the *location_service*. The following denial is written to the system log: *denied* `find` *on* `location_service` *to the* `ads_d` *domain*. As a result, the malicious component is terminated by the *ActivityTaskManager*.

The Ad library was included in the app as an *.aar* archive. To confine it, no modification was necessary, only the use of `AndroidManifest.xml` and `sepolicy.cil` was required.

A.3 Use case 3

In this use case we show how to confine a set of components, which rely on a high performance native library written in C to perform some task. Our goal is to demonstrate that the context running the native library code is prevented to access the network, even when the permissions *INTERNET* and *ACCESS_NETWORK_STATE* are granted to the app sandbox.

The native library is invoked by *UseCase3Activity*, which, according to line 4 in the `seapp_contexts`, is executed in a process labeled with `media_d` by *Zygote*. The call to the library is performed via JNI. Its job is to connect to the *camera_service* and take a picture. Since the app is granted the *CAMERA* permission, the native library code (legitimately, line 53 in the `sepolicy.cil`) connects to the `CameraManager`.

Since the native library performs image processing, we do not want it to access the network. However, the permissions *INTERNET* and *ACCESS_NETWORK_STATE* are granted to the app, as they are required by the Ads framework. Thus, when the policy module is missing, the native library can connect to the `ConnectivityManager` and successfully bind the current process to the network. Instead, when the policy module is enforced by SEApp, since `media_d` was granted only the basic app permissions (line 11 in `sepolicy.cil`), the

connection to the network is forbidden. This happens because binding a process to the network is associated with opening a network socket, an operation not permitted by SELinux without the required permissions. The following denial is written to the system log: *denied* `create` *on* `udp_socket` *to* `media_d` *domain*.

This use case, besides showing how SEApp confines a native library, also demonstrates the power and simplicity of the macro, as adding the line `(call md_netdomain (media_d))` to the policy module grants to `media_d` the needed permissions to access the network. The application developer is thus not required to know or understand the internal SELinux policy in order to leverage this functionality.

The isolation properties introduced by SEApp applies also to other common security problems presented in [51]. Just to mention one, SEApp can mitigate the impact of incorrect sandboxing of a scripting language.

A.4 Showcase app policy module

Here we report the showcase app policy module files.

```
user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.core_logic_d name=com.
  example.showcaseapp:core_logic levelFrom=all
user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.user_logic_d name=com.
  example.showcaseapp:user_logic levelFrom=all
user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.ads_d name=com.example.
  showcaseapp levelFrom=all
user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.media_d name=com.
  example.showcaseapp:media levelFrom=all
```

Listing 4: showcase app seapp_contexts

```
.*                  u:object_r:app_data_file:s0
files/confidential  u:object_r:
  com_example_showcaseapp.confidential_t:s0
files/ads_cache     u:object_r:
  com_example_showcaseapp.ads_t:s0
```

Listing 5: showcase app file_contexts

```
<?xml version="1.0" encoding="iso-8859-1"?>
<policy><signer signature="SIGNATURE">
<package name="com.example.showcaseapp">
<seinfo value="showcase_app"/></package>
</signer></policy>
```

Listing 6: showcase app mac_permissions.xml

```
(block com_example_showcaseapp
; creation of domain types
(type core_logic_d)
(call md_untrusteddomain (core_logic_d))
(type user_logic_d)
(call md_appdomain (user_logic_d))
(type ads_d)
(call md_appdomain (ads_d))
(call md_netdomain (ads_d))
(type media_d)
(call md_appdomain (media_d))
(typeattribute domains)
(typeattributeset domains (core_logic_d
  user_logic_d ads_d media_d))
; creation of file types
(type confidential_t)
(call mt_appdatafile (confidential_t))
(type ads_t)
(call mt_appdatafile (ads_t))
; bounding the domains and types
(typebounds untrusted_app core_logic_d)
(typebounds untrusted_app user_logic_d)
(typebounds untrusted_app ads_d)
(typebounds untrusted_app media_d)
(typebounds app_data_file confidential_t)
(typebounds app_data_file ads_t)
; grant core_logic_d access to confidential files
(allow core_logic_d confidential_t (dir (search
  write add_name)))
(allow core_logic_d confidential_t (file (create
  getattr open read write)))
; grant ads_d access to ads_cache files
(allow ads_d ads_t(dir(search write add_name)))
(allow ads_d ads_t(file(create getattr open read
  write)))
; minimum app_api_service subset
(allow domains activity_service (service_manager
  (find)))
(allow domains activity_task_service (
  service_manager (find)))
(allow domains ashmem_device_service (
  service_manager (find)))
(allow domains audio_service (service_manager (
  find)))
(allow domains surfaceflinger_service (
  service_manager (find)))
(allow domains gpu_service (service_manager (find
  )))
; grant core_logic_d the needed permissions
(allow core_logic_d restorecon_service (
  service_manager (find)))
(allow core_logic_d location_service (
  service_manage r(find)))
; grant ads_d access to unity3ads needed services
(allow ads_d radio_service (service_manager (find
  )))
(allow ads_d webviewupdate_service (
  service_manager (find)))
(allow ads_d autofill_service (service_manager (
  find)))
(allow ads_d clipboard_service (service_manager (
  find)))
(allow ads_d batterystats_service(service_manager
   (find)))
(allow ads_d batteryproperties_service (
  service_manager (find)))
(allow ads_d audioserver_service (service_manager
   (find)))
(allow ads_d mediaserver_service (service_manager
   (find)))
; grant media_d the needed permissions
(allow media_d autofill_service (service_manager
  (find)))
(allow media_d cameraserver_service (
  service_manager (find))))
```

Listing 7: showcase app sepolicy.cil