SAAC: Secure Android Application Context a Runtime Based Policy and its Architecture

Guillaume Averlant, Eric Alata, Mohamed Kaâniche, Vincent Nicomette, Yuxiao Mao

SAAC: Secure Android Application Context A Runtime Based Policy and its Architecture

Guillaume Averlant, Éric Alata, Mohamed Kaâniche, Vincent Nicomette and Yuxiao Mao
LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
Email: firstname.lastname@laas.fr

***Abstract*—A smartphone runtime environment consists of multiple entities with different goals and scope. Indeed, critical applications such as banking applications that make contactless payments, share the same environment with other applications of varying levels of trust. This paper presents a novel approach allowing a standard Android user to launch its applications in a configurable secure execution context. The security rules defined for each application are enforced by a dedicated security architecture implemented in several parts of the Android ecosystem. A performance assessment of the solution is also provided.**

I. Introduction

Smartphones are nowadays increasingly used to support several daily life activities, and more and more hold both personal and professional data. Furthermore, with the continuous development of wireless communication technologies, such as *Bluetooth Low Energy* (BLE) or *Near-Field Communication* (NFC), it is possible now to remotely control IoT devices or perform contactless payments with a smartphone. However, this large amount of data is generated by multiple applications of varying levels of trust, which raises serious concerns about the security and the privacy of the users.

Our work focuses on the Android platform, which already implements a permission system that controls access to system resources and restricts applications actions. Nevertheless, these permissions on their own are not adequate to provide the expected level of segregation, as they do not take into account the execution context. Indeed, it is not possible to define rules that constrain the permission system based on runtime observations. For example, it should be interesting from a security point of view to forbid any usage of the NFC component while a banking application is processing a contactless payment, so that other applications can not tamper with the process. Similar attacks have been demonstrated on BLE [1] and Bluetooth [2].

Standard isolation solutions used in *Bring Your Own Device* contexts, like *Android for Work* or *Samsung KNOX*, define a strong isolation between the personal and the professional environment. However, such solutions involve strong resources isolation that may not be optimal for the end users, as the resulting usability restrictions may not be acceptable for them.

In this paper, we present two contributions to tackle the aforementioned issues: i) A novel approach allowing a standard Android user to control applications execution rights or their access to system resources depending on the smartphone runtime context; ii) An isolation architecture, based on low-level application communications interception, and implemented in several parts of the Android ecosystem, which enforces these security rules.

The paper is organized as follows. Section II provides a short background of the Android software ecosystem, and Section III discusses the state of the art. Section IV presents the attack model and introduces the *secure manifest*, which is an additional manifest dedicated to store our security rules. Section V outlines the architectural details of our isolation framework which is then evaluated in Section VI.

II. Android background

Android is a fully fledged and open source OS developed by Google under the *Android Open Source Project* (AOSP). It essentially consists of a framework layer that runs under the supervision of a customized Linux kernel. This layer drives the execution of *Android applications* (so-called "apps" in the remaining of the paper). Android apps are written in Java and are executed under a specific runtime environment, named *Android Runtime* (ART). Additionally, Android apps can embed native code parts written in C/C+ for shared libraries or to support high performance computing. Specifically, Android apps can be decomposed into four types of *components*:

- Activities: represent the different screens of the apps.
- Services: handle background tasks.
- Content providers: allow to share data with other apps.
- Broadcast receivers: manage asynchronous events.

Android apps rely on the framework which provides a rich set of APIs for both Java and native languages. Additionally, the framework includes numerous system services and daemons dedicated to manage apps lifecycle and system resources, and embodies an Inter Process Communication (IPC) system, named *binder*. This communication channel allows application components to interact with each other or with system services seamlessly, in the same way as if they were performing *Remote Procedure Calls* (RPCs). The binder subsystem communicates with the binder driver included in the Linux kernel through *ioctl* system calls. Furthermore, this IPC mechanism is also used to carry higher level communication messages named *Intents*. The latter represent an abstract description of an operation, either implicit or explicit, and are used to start activities/services or broadcast events. Intents are handled by the *Activity Manager Service* (AMS).

Other more privileged layers are also parts of the Android ecosystem. Android being mostly developed for the ARM CPU architecture, the details provided are specific to this

architecture. These most privileged layers are enabled by several extensions included in current ARM processors. One of them, the virtualization extensions, bring up the ability to execute a hypervisor at a higher privilege level than the Linux kernel, and to benefit from virtualization capabilities.

Furthermore, an existing security model is deployed in several parts of Android. Firstly, a sandboxing mechanism isolates apps from each others and from other parts of the system. All apps are launched in a different process, thus benefiting from the Linux process isolation, and are restricted in their file access capabilities. Indeed, a Linux *User ID* (UID) is assigned to each installed app, thus ensuring that they can only access their own files or world readable ones. Consequently, Android apps must use the Binder IPC mechanism or emit direct system calls to the kernel to communicate with other parts of the system. Secondly, an app must have an adequate access right in order to access a specific resource. Those access rights, named *permissions*, are either declared by an app at install time in their manifest, or requested from the user at runtime since Android 6.0. Each resource access is then enforced in different parts of the Android ecosystem, such as in system services or daemons.

III. State of the art

A. Access control policies

Several access control models for Android have been proposed. Most notably Guo et al. [3] defined a security framework composed of both *Mandatory Access Control* (MAC) and *Role Based Access Control* (RBAC) models for Android. Additionally, the work of Smalley et al. [4], which ported the SeLinux MAC model to Android, has been integrated into the Android code-base since the 4.3 version.

Some context-aware policies have been studied in the past. In [5], [6] spatial or temporal contexts, determined from the hardware sensors, are used to define the current set of security rules enforced for each running application. DR BACA [7] and CA-ARBAC [8] focused on integrating context-awareness in RBAC models: A role, carrying a set of allowed permissions, is associated to each application. Then, either a new role is assigned to each application when the context changes [7], or the set of permissions associated to the role are conditionally enabled depending on the current context [8].

Another study [9] has considered the use of machine learning and context sensing techniques to dynamically select the access control rules for each application based on the current context. The main advantage is to reduce the burden of the policy definition and configuration by a standard end user, which is assumed to not have the required expertise to perform this task. The usability concerns have been also investigated in [10], where machine learning is used to create context based access control rules that match the user privacy preferences. Furthermore, resources can be obfuscated instead of only allowing/denying their access, therefore lowering the chance of application crashes while preserving some privacy.

Most of the aforementioned works have only considered external context sources (such as Geolocation) and the global phone state but none of them (except [10]) took into account the list of running applications as part of their context definition. Furthermore, in this paper we also aim at providing a solution usable and configurable by ”non experts” Android users but without using machine learning techniques.

B. Isolation architecture

As stated in our previous work [11], an efficient isolation for Android applications should rely on a multi-level architecture, providing the required control over all communication means, while avoiding any semantic gap and still being resilient to low levels attacks. To our best knowledge, such type of architecture has not been addressed in the previous research. Furthermore, many access control model implementations do not even consider the communication paths accessible by native code embedded in applications. Indeed, their interceptions only rely on either the framework modification [5]–[7] or on its dynamic instrumentation [10]. The remaining implementations are based on both framework modifications together with kernel level modifications. Especially, [9] is based on the Flaskdroid framework [12] which uses the existing Linux kernel hooks provided by SEAndroid [4], and adds several hook points in the Android framework called *Userspace Object Managers*. Similarly, CA-ARBAC [8] uses the *Android Security Modules* (ASM) [13] that provide the required kernel and framework hooks. In addition, SEAndroid and ASM take advantage of *Linux Security Modules* (LSM) framework which provides hooks in different parts of the Linux kernel.

IV. Attack model & secure manifest

Our solution aims to describe and enforce several rules that complement those defined by the existing permission system in Android. These rules specify additional access restrictions to resources. They are defined in supplementary *secure manifests* that are specific to each application and can be complemented by user input. There are two types of rules: static rules which further restrict the resource access of the target application; dynamic rules, or *secure context*, which limit the access of other applications to shared resources while the target application, or one of its components, is running.

In our attack model, the attack source is an untrusted application installed by the user and which can be remotely controlled by the attacker. This application can either be a malevolent app or a benign app hijacked by the attacker. The goal of this app is to bypass the rules defined in its secure manifest, or those of other installed apps.

We can identify several types of applications that could be targeted by any other application controlled by an attacker. For example, a social media app, or any application with a login page, could be hijacked to retrieve the login credentials and therefore steal the identity of the smartphone user. Any application that performs online or contactless payments could be also subject to money robbery. Furthermore, if the user does not need all the features of an application, he can restrict resource access of an application to preserve his privacy.

To illustrate the capabilities extent of the secure manifest, we go through an example provided in the Listing below.

```
<secure_manifest>
  <pkg_info name="com.example.app" pub_key="..." />
  <!-- Resource restriction for the package -->
  <resource_restriction>
    <resource name="camera" />
  </resource_restriction>
  <!-- Secure context for an app component -->
  <secure_context target="login_activity">
    <!-- restrictions for the following apps -->
    <app_restriction>
      <app_list>
        <app name="com.example.badapp" />
      </app_list>
      <resource_restriction>
        <resource name="wifi" />
      </resource_restriction>
    </app_restriction>
    <!-- restrictions for all running apps -->
    <resource_restriction>
      <resource name="nfc" />
    </resource_restriction>
  </secure_context>
</secure_manifest>
```

The first part of the manifest is dedicated to application information, like the package name and the developer public key. This kind of data is used to match the Android app with its secure manifest. Furthermore, an alert can be raised when the public key mismatches the one used to sign the app.

The second part is dedicated to the static rules and defines the list of resources banned for the app. When the target app tries to access these resources, the policy handler service replies with empty data. Therefore, the application should not crash due to the unavailability of a required resource.

The third part is related to the dynamic rules, or secure context rules, which define resource access control when one or more components of the application are running. We designed this part to cover the maximum of cases: The secure context target can be as large as the whole application, or limited to one or a list of components separated by commas. Furthermore, each secure context definition is additive. More precisely, if two secure contexts are defined, one for a specific component and one for the whole app, if this specific component is running, the policy handler service aggregates all dynamic rules defined in both secure contexts. In addition, the secure context can specify rules that further restrict the resource access of a list of applications, or just prohibit the use of these applications if no resources are mentioned.

For Android, a resource can either refer to the ability to access data or to perform an action on the system, like enabling the wifi. In the secure manifest, we consider as for now 20 resources gathered into 6 groups, listed in Table I.

Table I - CONSIDERED RESOURCES

Group	Resources
Location	Coarse location, GPS
Communication	Mobile data, Wifi, Bluetooth, NFC
Peripherals	Camera, Microphone, Motion sensors, Environmental sensors
Personal data	SMS, Contacts, Phone, ID information
Storage	Internal storage, SDCard, USB devices
High risks resource	System settings, Draw over applications, Automation services (e.g. accessibility...)

Motivating examples

1) Banking application: A banking application is a critical application and a user may wish to run such an application in a paranoid mode, i.e., no other application is authorized to run at the same time. This can be used to mitigate attacks such as Cloak and Dagger [14]. This case corresponds to a simple `<secure_context>` containing an empty `<app_restriction>`.

2) Mutual exclusion of applications: A user may wish to never run at the same time two specific applications. For instance, an application collecting private data (such as a fitness app that may collect private health and location data) and facebook app. To do this, the `<secure_context>` must include an `<app_restriction>` containing a single `<app />` tag for the facebook app.

3) Mutual exclusion of a specific resource: The mutual exclusion may be specified at the granularity of a specific resource usage. For instance, a user may forbid the use of the NFC device by any other application when it is used by his banking application during a payment operation. In this case, a single `<resource_restriction>` containing the "nfc" `<resource />` must be provided for the `<secure_context>` targeting the payment operation.

V. SECURITY ARCHITECTURE

To enforce the secure manifest rules, we designed a security architecture able to intercept the required communication paths between applications and resources. The architecture is built around the following design goals: **G1** *Complete application communication mediation.* All possible application communication channels shall be intercepted to ensure a thorough policy enforcement. **G2** *Integrity protection.* The integrity of our solution must be protected to ensure its robustness against attacks. **G3** *Limited changes over the existing codebase.* To help implementing our solution in future Android versions, the amount of modifications over the existing Android code must be reduced. **G4** *Efficiency & usability.* Also, the performance impact must be as low as possible. Furthermore, the mandatory interactions with the user should be limited for better usability.

A. Components overview

To fulfill the design goals, the security architecture is split into four different components. This architecture embodies a multi-level isolation approach for whom a preliminary concept has been presented in [11]. Figure 1 recaps the components of our solution.

The first component, named *policy handler service* (PHC), is the keystone that links the other components of our security architecture together. It includes most of the control logic that manages the secure manifest of each hosted application. Furthermore, it aims at intercepting high level IPC initiated by running apps, namely the *Intent* messages (**G1**). For this purpose, this component is implemented as a system service of the framework, tightly coupled with the AMS. To be

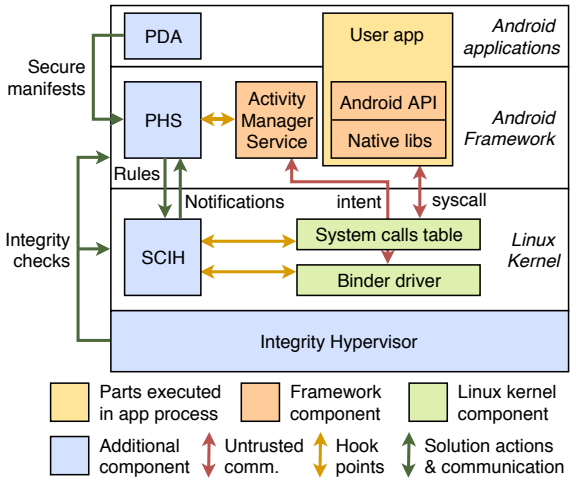

Figure 1. Solution overview

more specific, we added several hook points in the AMS to mediate the Intent execution, as well as to retrieve information about the running application components states. That way, we are able to be aware of the applications life-cycle which is mandatory to enforce the policy.

The second component, named *system call interception handler* (SCIH), aims at intercepting the remaining communication channels of applications, i.e. lower-level binder transactions and system calls (**G1**). We implemented this component as a Linux kernel module which hooks the system calls table to mediate the aforementioned communication channels. We decided not to use the LSM hooks to avoid interfering with the SELinux subsystem, as well as to keep more freedom over the intercepted data. Furthermore, the module performs dynamic memory introspection in the binder driver to identify the target of a binder transaction (**G3**). In addition, the module implements a basic character device which allows the communication between the SCIH and the PHS through ioctl system calls.

The third component, named *integrity hypervisor* (IH), is a bare metal hypervisor which acts as the root of trust of our solution (**G2**). It performs integrity checks over critical parts of the Linux kernel to detect any malicious alteration [15], traps accesses to crucial kernel memory pages, and checks at startup the signature of the other parts of our solution.

The last component, the *policy definition application* (PDA), is a standard system application providing the user interface of our solution. It aims at: 1) retrieving the default secure manifest of each installed application; 2) allowing the user to modify the secure manifests by adding/removing rules; 3) appropriately inform the user about any conflicts over the current policy at runtime (**G4**). The PDA communicates with the PHS with traditional binder transactions.

B. The architecture in details

When a user installs a new application, the PDA retrieves the corresponding default manifest automatically and stores them in a specific private folder that is therefore not accessible by other applications (**G2**). The PHS then retrieves and parses the manifest at startup or when any modification on the folder occurs. In order to generate the rules from the manifest, the PHS needs to convert several strings into meaningful information. It has to: 1) match each application ID with it corresponding Linux user id; 2) associate each system resource name to the set of object identifiers that provide the resource.

Once the rules are generated from the retrieved data, the PHS has to decide when to enforce the rules associated with each app components. To do this, it needs to be aware of the application components lifecycle, i.e. when the components are started or stopped. There are multiple cases. Activities can be started either by intents, as a result of pressing the ”back” key, or by using the *recent apps* screen. These events can be intercepted by inserting additional hooks in the AMS. The same kind of hooks can be applied for services and content providers, as services are only launched via intents and content providers are directly managed by the AMS. However there are no similar ways to intercept the component shutdown, as processes running these components can be killed at any moment due to low memory. Therefore the PHS has to monitor the AMS internal structures to detect this kind of event.

Our architecture must also handle potential conflicts that may occur at runtime, which are actions, forbidden by the current rules, that an application tries to perform. They can either result from an operation that the user wants to achieve, or not at all. Therefore, we are not able to resolve the conflicts automatically, because this should depend on the user decision. To do this, the conflict details are displayed to the user, and he is asked to allow or deny the action. For the first case, all applications in conflict with the action to be performed have to be closed. Otherwise, the operation is blocked. The conflicts are either detected by the SCIH or the PHS depending on the kind of access. In both cases, we first have to block the access, and then ask the user. Otherwise, the source application would hang up until the conflict resolution, and be considered as faulty by the system. Then, after the conflict resolution, the same operation is allowed (or not) to be executed. The user interaction for conflict resolution is carried out by the PDA, which customizes its UI depending on the state of the operation issuer. If it is a foreground activity, the user is asked to take a decision via a popup. Otherwise, only a notification is issued by the PDA, allowing the user to answer whenever he wants (**G4**).

VI. Implementation & Early Performance Results

Our implementation is based on the Android 8.1 source code available on the *Android Open Source Project* (AOSP) repositories. Two kinds of hardware are used: i) A computer running the x86_64 version of the Android emulator, with a quad-core Intel Xeon E3-1270 clocked at 3.6GHz and running the Ubuntu 17.10 OS with 16GB of RAM; ii) The *96boards hikey* development board equipped with an octa-core ARM Cortex-A53 64-bit clocked at 1.2GHz and 2GB of RAM.

Table II - OVERHEAD RESULTS

	intent intercept. (3000 rules)	intent intercept. (15000 rules)	periodic check (7 running apps)	periodic check (14 running apps)	PHS ⇔ SCIH comm.	PHS ⇔ SCIH comm. w/ syscalls intercept.
1st quartile	0.639ms	1.923ms	0.525ms	0.678ms	1.409ms	1.609ms
median	0.672ms	1.989ms	0.612ms	0.931ms	1.590ms	1.776ms
3rd quartile	0.740ms	2.081ms	0.843ms	1.050ms	2.136ms	2.285ms

We have implemented a prototype of a bare-metal hypervisor on the hikey development board. Indeed, as the board bootloader is mostly open source, we used the virtualization capabilities of the CPU for this purpose. The rest of the implementation has been performed and tested on the Android emulator, and only the PDA and the SCIH are not yet fully featured. Nevertheless, it is possible to evaluate the overhead of our interception solution, as all the corresponding code has been developed. All tests performed were carried out on the Android emulator, and are gathered in Table II.

We first focused on the performance of *intents* interception by the PHS. Without any modifications, the system performs an intent delivery in about 30±10ms (resp. 65±25ms if the destination app is not running). With the PHS interception enabled, the overhead is around 672µs for 3000 rules, and 1.989ms for 15000 rules, including both the interception and the rules evaluation time. Even with the extreme number of 15000 rules, this only corresponds to a 6.6% overhead. We also assessed the overhead of component termination checks performed by the PHS. This operation is performed every 500ms, and its execution time fluctuates in accordance with the number of app launched. For instance, this operation takes 612µs with 6 apps running, and 931µs with 14. Finally, we measured the communication time between the PHS and the SCIH, that occurs for each rule to be enforced or removed by the SCIH when an app component starts or stops, or when the SCIH must notify the PHS about a conflict. Our analysis shows that a round trip between the PHS and the SCIH incurs a delay of 1.590ms. Indeed, this delay is mainly caused by the context switch between the framework and kernel. Although we estimate that this specific delay should cause most of the overhead induced by our security solution, there is always room for optimization. For instance, we can gather multiple rules in one message to reduce the resulting overhead. Furthermore, we performed an analysis of the overhead induced by the system call interceptions. To do this, we compared standard communications between the PHS and the SCIH with and without the aforementioned interception. The resulting overhead is about 200µs. Globally, the measured interception overheads are significantly low and should not lead to any impact on the user perceived performance.

VII. PERSPECTIVES

As ongoing work, we are aiming at finishing the implementation of a fully integrated prototype of our solution. In the future, we plan to focus on the retrieval of default secure manifest for each application through crowd sourcing. Another point that we plan to address is the adaptation of the isolation behavior of our solution based on the resource type. Indeed, nonvolatile data like the one present in the SDCard cannot be mediated the same way as volatile data produced by in-device sensors. Furthermore, we seek to provide an extension to the secure manifest that would allow finer grain resources parameters to the policy. For instance, a user may want to restrict the Internet access of some applications by defining a white-list of allowed URL.

REFERENCES

[1] P. Sivakumaran and J. Blasco, "Attacks Against BLE Devices by Co-located Mobile Applications," *ArXiv E-Prints*, Aug. 2018.

[2] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside Job: Understanding and Mitigating the Threat of External Device Mis-Bonding on Android," in *Proceedings 2014 Network and Distributed System Security Symposium*, 2014.

[3] T. Guo, P. Zhang, H. Liang, and S. Shao, "Enforcing Multiple Security Policies for Android System," in *2nd International Symposium on Computer, Communication, Control and Automation*, Apr. 2013.

[4] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android." in *NDSS*, vol. 310, 2013.

[5] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "CRêPE: A System for Enforcing Fine-Grained Context-Related Policies on Android," *IEEE Trans. Inf. Forensics Secur.*, vol. 7, no. 5, Oct. 2012.

[6] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes, "MOSES: Supporting and Enforcing Security Profiles on Smartphones," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 3, May 2014.

[7] F. Rohrer, Y. Zhang, L. Chitkushev, and T. Zlateva, "DR BACA: Dynamic Role Based Access Control for Android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[8] J. Abdella, M. Özuysal, and E. Tomur, "CA-ARBAC: Privacy preserving using context-aware role-based access control on Android permission system," *Secur. Commun. Netw.*, vol. 9, no. 18, Jan. 2017.

[9] M. Miettinen, S. Heuser, W. Kronz, A.-R. Sadeghi, and N. Asokan, "ConXsense: Automated Context Classification for Context-aware Access Control," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, 2014.

[10] K. Olejnik, I. Dacosta, J. Soares Machado, K. Huguenin, M. E. Khan, and J.-P. Hubaux, "SmarPer: Context-Aware and Automatic Runtime-Permissions for Mobile Devices," in *38th IEEE Symposium on Security and Privacy (S&P)*, May 2017.

[11] G. Averlant, "Multi-level Isolation for Android Applications," in *Software Reliability Engineering Workshops (ISSREW), 2017 IEEE International Symposium On*, Oct. 2017.

[12] S. Bugiel, S. Heuser, A.-r. Sadeghi, and T. U. Darmstadt, "Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware," Intel Collaborative Research Institute for Secure Computing, Tech. Rep., 2012.

[13] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "ASM: A Programmable Interface for Extending Android Security," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[14] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2017.

[15] B. Morgan, E. Alata, V. Nicomette, M. Kâaniche, and G. Averlant, "Design and Implementation of a Hardware Assisted Security Architecture for Software Integrity Monitoring," in *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2015.