Multi-level Isolation for Android Applications

Guillaume Averlant

Multi-level isolation for Android applications

Guillaume Averlant
LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
Email: firstname.lastname@laas.fr

***Abstract*—Android is one of the most popular operating systems on mobile devices, and its usage is not going to decrease anytime soon. Although Android security has already been widely studied in the literature, its quick evolution scheme and emerging usages call for the continuation of this common effort.**

In particular, mobile devices are now commonly used in different contexts, like in a *bring-your-own device* (BYOD) environment where personal and business data are held on the same device. However, small and medium companies cannot afford the costs of a dedicated software solution allowing personal and professional data to securely coexist on employees devices.

In this paper, we present a preliminary solution to address this need of isolation by leveraging hardware virtualization extensions found in current mobile processors. This solution targets chipset manufacturers and provides security and privacy protections for standard Android end users.

INTRODUCTION

Android is one of the most popular operating systems on mobile devices, and its usage is not expected to decrease in the near future. Although Android security has already been studied in the literature, the continuous evolution of its implementation and of processors technologies and the new emerging usages call for the continuation of this common effort to address open challenges, e.g., as regards the need to ensure the proper isolation of the execution of multiple applications with different security requirements.

In particular, mobile devices are now commonly used in different contexts, like in a *bring-your-own device* (BYOD) environment where personal and business data are held on the same device. However, small and medium companies cannot afford the costs of a dedicated software solution allowing personal and professional data to securely coexist on employees devices.

Even for personal usage, an isolation layer could be employed to segregate the data from different contexts between each other. For example, one may want to isolate its banking application from the rest of the system.

However strong isolation may not be the optimal solution, because the resulting usability restrictions may not be acceptable for the end users. In this purpose, we envision a controlled permeability between isolation layers. For example, this permeability can be used to retrieve contact information from an isolated application; or even to allow an isolated application to provide its services to other applications.

With the evolution of mobile processors, technologies like virtualization that were reserved to their desktop/server counterparts are now available for mobile applications. These new technologies offer opportunities to design new isolation solutions or complement existing ones.

One possible solution could be to rely on the hypervisor to provide isolation at application level. Nevertheless, due to the semantic gap, this solution is not recommended. Instead, we promote the use of a multi-level isolation solution, implemented in different parts of the Android ecosystem.

In this paper, we present a preliminary solution to address this need of isolation by leveraging hardware virtualization extensions that are now available in current mobile processors. This solution targets chipset manufacturers and provides security and privacy protections for standard Android end users. The paper is organized as follows. Section I introduces the Android software ecosystem. Then, Section II discusses the state of the art. Finally, Section III outlines the proposed solution.

I. THE ANDROID ECOSYSTEM

The techniques used to provide an isolated environment for Android applications can be implemented at several levels in the Android ecosystem stack. In order to analyze the different solutions provided in the literature, we first introduce the Android environment.

Android OS, developed by Google, is mainly composed by a framework layer on top of a customized Linux kernel. This layer, named *Android Application Framework* (AAF), drives the execution of Android application [1]. It consists of multiple parts [1]:

- A set of APIs provided to applications.
- An application management system.
- A permission management system [2].
- An Inter-Components Communication system [3].

APIs provided to Android applications are available in two forms: the Java API and the Android NDK. The latter provides an API mapping for parts of applications written in native language, as well as a custom libc named bionic.

The application management system handles applications installation, update and removal. Moreover, it checks application signatures when they are installed and updated in order to ensure their integrity.

The permission management system ensures that each application satisfies the set of permissions allowed by the end user. These permissions are either defined in the application manifest and accepted at install time, or requested at runtime.

The Inter-Components Communication system enables the communication between different applications or system services. These communications are mediated by a custom *Inter-*

[1] Written in Java or native-code languages such as C and C++

Process Communication (IPC) system: a kernel component named Binder which is specific to Android OS.

Furthermore, the Android ecosystem definition is not restricted to the Android OS; it basically consists of the whole software stack running on top of the ARM processor, and leveraging from its inherent capabilities.

More precisely, several extensions included in current ARM processors allow the device maker to insert other layers to this ecosystem:

- 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.
- The security extensions, also known as TrustZone, split the execution environment into two worlds. The secure world, isolated from the normal world, is able to host a secure environment where the device maker can run applications and services [2].

These technologies are notably used by state-of-the art solutions introduced later in this paper, as well as by existing devices [4].

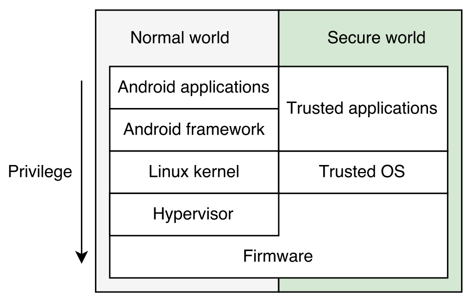

Figure 1. Android ecosystem - a global view

Figure 1 describes the different layers of this ecosystem. This Android ecosystem overview is needed to better understand the scope and the target of the existing isolation solutions defined in the literature which are presented in the next section.

II. State of the Art

Many improvements to the Android ecosystem security have been proposed. They address several imperfections in protecting the end-user security and privacy. These solutions can be classified into multiple types, which have distinctive goals, and affect different parts of the ecosystem:

- Kernel hardening and kernel integrity enforcement tools: Aim at providing mitigation techniques, or detecting any malicious modification of the Linux kernel.
- Data flow analysis and information flow tracking techniques: Enable the system to evaluate the evolution of applications data and their sharing with other applications. They are notably useful to track privacy violations.
- Application behavior analysis solutions: Evaluate the behavior of installed applications and compare them to an established reference model. This model can be created by using machine learning techniques.
- Sandboxing or containerization techniques: Can be implemented at different levels of the Android ecosystem. They isolate different entities of these levels from each other. This isolation is controlled by a reference monitor which implements an access control policy.

In this paper, we focus on containerization techniques. So, in the following subsections, we present existing sandboxing solutions found in the literature organized according to their implementation origin in the Android ecosystem stack. Depending of this origin, the range of action and the manipulated data of these solutions might be quite different.

A. System-level containerization

System-level containerization solutions aim at providing multiple isolated environments by leveraging from the virtualization and security extensions of recent ARM processors. Their implementation is based on a hypervisor providing virtual environments to several isolated Android OS. Their execution is thus controlled by this hypervisor which acts as the reference monitor.

The work of Lengyel et al. [5] uses a bare metal Xen hypervisor to provide this virtualization layer, and employs several Xen modules to enforce its security. Furthermore, a TrustZone component checks the hypervisor integrity at startup and runtime. It also uses *Virtual Machine Introspection* (VMI) to control the runtime integrity of the underlying OS.

Another goal may also be the isolation of security critical components from the rest of the Android OS. An obvious solution is to implement these components in the secure world provided by the security extensions. However this approach enlarges the system *Trusted Computing Base* (TCB) by increasing the trusted code size. This problem is for example addressed by Cho et al. [6] and Sun et al. [7].

B. OS-level containerization

OS-level containerization techniques intend to propose the execution of isolated groups of applications having different access privileges to system resources. This isolation is either provided by Android framework virtualization, or by applying fine grained policy on framework services and user applications to restrict their access to system resources.

A prime example is Condroid [8] which leverages the cgroup feature of the Linux kernel to define namespace based isolation between several Android framework instances.

In comparison, the work of Fernandes et al. [9] focuses on isolating a group of trusted applications from untrusted ones. This isolation is achieved by deprivileging several parts of Linux kernel and the Android framework.

Before the 4.3 version, Android only used the *Discretionary Access Control* (DAC) isolation policy of Linux by assigning an UID to each application. Since then, SeLinux implementation for Android [10] has been integrated into the Android codebase; SeLinux being a Linux Security Module providing a MAC policy to userland application, it has been customized to integrate the peculiarities of Android OS [3].

[2]E.g. Fingerprint authentication services, or DRM management applications

[3]E.g. Multiple SeLinux hooks have been integrated to the binder

Inspired by SeLinux design but with a more important integration in the Android framework, ASM [11] introduced an API to implement user defined security modules: The MAC policy is loaded from standard Android application.

C. Framework-level containerization

Framework-level containerization solutions have almost the same goal as OS-level containerization techniques. But instead of modifying the Linux kernel code, they rely on isolation solutions implemented in the Android Framework.

For example, with Pinpoint [12], Ratazzi et Al. modified the binder architecture to implement namespace support [4]. With this new concept, they are able to protect sensitive resources (contacts, IMEI, location...): Applications are associated to namespaces that dynamically assign them real resources or user controlled stubs.

As stated in the work of Neuner et al. [13], an information flow tracking framework, named Taintdroid [14], is used in numerous sandboxing solutions. However, its implementation relies on Dalvik VM which is no longer used in Android (since Android 5.0) and was replaced by ART compiler. Its successor, TaintART [15], provides taint tracking through the modification of this new ART compiler as well as the binder. New sandboxing solutions could leverage information flow tracking from TaintART to implement their isolation policy.

D. Application-level containerization

Application-level containerization techniques are dedicated to provide user controlled isolation solutions and do not require any modification of the Android OS. They always consist of a monitoring application which mediates all resources access from one or more isolated applications.

In Boxify [16], this goal is achieved by launching the isolated applications in an unpriviledged process. All binder IPC and other system calls are then trapped in the monitoring application, which is able to apply a user defined policy.

A different solution is proposed by Bianchi et Al. [17]. This time, a stub application is generated for one or more isolated applications. The latter are launched by a stub process which catches their system calls using a syscall interposition technique based on ptrace.

Through the study of all these containerization solutions, we can observe that: The more privileged the solution, either the less fine grained is the resulting policy, or the more effort should be employed to fill the semantic gap between the isolation target and the monitoring software. For example, a hypervisor isolating Android applications with fine grained policy should implement a complex VMI logic to gather the required data; and this complex logic increases its attack surface. However, being a high privileged software increases the isolation effectiveness by preventing the target to bypass the resulting isolation. Based on these statements, we propose the design of a new solution in the next section.

[4]Which embodies the concept of Linux Namespace lightweight isolation

III. PROPOSED SOLUTION

In the previous section, we observed that implementing an isolation solution at a higher or lower privilege level has an impact on the type of data that can be accessed. In this section, we introduce our proposed solution designed to provide application isolation to Android end users. This solution targets chipset makers because we need to modify the most privileged layers of the Android ecosystem.

Our solution is designed to ensure a sufficient isolation level to protect the end user against the following scenarios:

- A malevolent application attempting to exploit system vulnerabilities to perform a system privilege escalation.
- A malevolent application trying to steal sensitive data.
- A non malevolent application used as an attack vector.
- A non malevolent application whose privacy policy is not suitable for the end user.

Figure 2 describes our proposed solution architecture. It embodies a multi-level design to cope with the weaknesses of single-level solutions identified in section II.

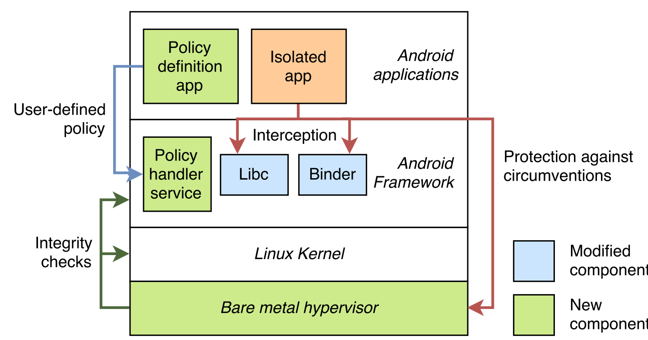

Figure 2. Proposed solution architecture

The implementation of this solution is based on the Android Framework. First, instrumenting the binder enables us to intercept binder IPC communications. Likewise, other syscalls are captured by inserting hooks in the libc. Finally, a dedicated service ensures the enforcement of the policies defined by the end user. This kind of IPC interception has already been achieved in existing solutions in the literature [11] and is a necessary step to apply user-defined policy to applications.

A second important part of our architecture is implemented as a bare metal hypervisor controlling the execution of the Linux kernel. Only a few functionalities are implemented in this hypervisor to keep its footprint as light as possible: It ensures the integrity of other parts of our solution, as well as critical components in the Linux kernel; and it checks the syscall sources to prevent any attempt to bypass the provided isolation. Indeed, an application may emit syscalls by their own without using our modified libc.

These checks could have been directly implemented in the Linux kernel but we choose to implement the most privileged part of our solution in a separate hypervisor instead for multiple reasons. First Linux is quite heavy (∼18M+ loc) and has inherently a non negligible attack surface. In addition, the hypervisor enables us to check the integrity of the kernel. Furthermore, thanks to the RISC concept of ARM architecture, the overhead cost due to the hypervisor should be lower than in x86 architecture. Finally, it is possible to use second level page table in a hypervisor to tag trusted pages where syscalls are allowed, in an efficient manner: This page table, that is used to provide an additional translation between the OS and the physical address space, contains a user-defined field which can be used to store this kind of information.

The last part consists of a specific Android application which allows the end user to define the isolation policy. Its integrity is protected by a cryptographic signature which is checked by the hypervisor at runtime. However, as real end users are rarely security specialists, we propose to retrieve a default security rules set for each installed application through a crowd-sourcing website. To complement these default rules, all newly installed applications go through a learning phase where each of its new communication with other applications should be approved by the user. These rules can then be modified later via the application UI. All rules are stored in a secure area only accessible by the policy handler service which handles their modifications: the policy definition application has no direct access to these rules.

To be more specific about the security rules, we want to take advantage of the high level of details concerning applications IPCs and system resources access retrieved by our hooks implemented in the Android Framework to allow the end user to define a fine grained security policy. User should be able to restrict any kind of IPC sent or received by an application, as well as restricting any access to system resources. For example, user should be able to restrict Internet access granted to an application to a subset of IP addresses or url. However, some applications may stop working after applying these restrictions on resources access. To address this issue, we need to be able to provide empty or fake information instead, like in [12].

IV. Conclusion, ongoing work and milestones

In this paper, we presented the principles of a multi-level isolation solution allowing Android end users to define fine grained security policy for installed applications. This solution leverages hardware virtualization extensions that are now available in current mobile processors, and therefore targets chipset manufacturers. Our approach is based on a multi-level architecture in which the policy's control logic is close to the isolated resources to reduce the semantic gap, while we benefit from a high privileged software to ensure the effectiveness and integrity of our solution. We have already experimented the feasibility of implementing a hypervisor on the *96Boards hikey* development board, and are now implementing our proposed solution. Currently, the hypervisor implementation is already underway, while the Android framework modifications are planned for September. We aim to submit a full paper exposing our research on this subject by the end of 2017, and my thesis defence is expected for the end of 2018.

Acknowledgment

My thesis is carried out at the Dependable Computing and Fault Tolerance research group at LAAS-CNRS, supervised by Éric Alata, Vincent Nicomette and Mohamed Kaâniche.

References

[1] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, S. Lee, and T. Kim, "Toward Engineering a Secure Android Ecosystem: A Survey of Existing Techniques," *ACM Comput. Surv.*, vol. 49, no. 2, 2016.

[2] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[3] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis," in *Presented as Part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, 2013.

[4] U. Kanonov and A. Wool, "Secure Containers in Android: The Samsung KNOX Case Study," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '16. ACM, 2016.

[5] T. K. Lengyel, T. Kittel, J. Pfoh, and C. Eckert, "Multi-tiered Security Architecture for ARM via the Virtualization and Security Extensions," in *2014 25th International Workshop on Database and Expert Systems Applications*, 2014.

[6] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016.

[7] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[8] W. Xu and Y. Fu, "Own Your Android! Yet Another Universal Root," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, 2015.

[9] E. Fernandes, A. Aluri, A. Crowell, and A. Prakash, "Decomposable Trust for Android Applications," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

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

[11] 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)*. USENIX Association, 2014.

[12] P. Ratazzi, A. Bommisetti, N. Ji, and W. Du, "PINPOINT: Efficient and Effective Resource Isolation for Mobile Security and Privacy," in *Proceedings of the SPW Workshop on Mobile Security Technologies (MoST)*, 2015.

[13] S. Neuner, V. Van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl, "Enter sandbox: Android sandbox comparison," in *Proceedings of the Third Workshop on Mobile Security Technologies (MoST)*, 2014.

[14] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 2014.

[15] M. Sun, T. Wei, and J. C. Lui, "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. ACM, 2016.

[16] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged App Sandboxing for Stock Android," in *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015.

[17] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "NJAS: Sandboxing Unmodified Applications in Non-rooted Devices Running Stock Android," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '15. ACM, 2015.