Improving Mobile Device Security with Operating System-Level Virtualization

Sascha Wessel, Frederic Stumpf, Ilja Herdt, and Claudia Eckert

Fraunhofer Research Institution AISEC, Munich, Germany
{sascha.wessel,frederic.stumpf,ilja.herdt,
claudia.eckert}@aisec.fraunhofer.de

Abstract. In this paper, we propose a lightweight mechanism to isolate one or more Android userland instances from a trustworthy and secure entity. This entity controls and manages the Android instances and provides an interface for remote administration and management of the device and its software. Our approach includes several security extensions for secure network access, integrity protection of data on storage devices, and secure access to the touchscreen. Our implementation requires only minimal modification to the software stack of a typical Android-based smartphone, which allows easy porting to other devices when compared to other virtualization techniques. Practical tests show the feasibility of our approach regarding runtime overhead and battery lifetime impact.

1 Introduction

Smartphones are already an omnipresent part of our everyday lives. They are used for various tasks with different security requirements like web browsing, banking, or business use cases. This results in an increased demand for isolated environments with different security levels for different tasks on a single device. Payment service providers want a secure environment to protect their applications for financial transactions. Companies want a corporate environment isolated from the private environment of a user and the possibility to manage the devices remotely. This especially includes the enforcement of various security policies, which cannot be enforced with a stock Android-based smartphone today, e.g., whitelisting and/or blacklisting of applications and versions of applications in case of known vulnerabilities.

In this contribution, we propose a lightweight isolation mechanism for Android based on operating system-level virtualization and access control policies to separate one or more Android userland instances from a trustworthy and secure environment. Furthermore, we propose several security extensions based on this environment to control and manage the Android instances and their input and output data. This includes secure network communication, integrity protection of data on storage devices, and secure access to the touchscreen, e.g., for password entry dialogs. Another important part of our security concept is the integration of a secure element (SE) (e.g., embedded into a microSD card) to store secret keys and data physically separated from the application processor of

L.J. Janczewski, H.B. Wolfe, and S. Shenoi (Eds.): SEC 2013, IFIP AICT 405, pp. 148–161, 2013.
© IFIP International Federation for Information Processing 2013

the smartphone. This is to ensure its protection even in case of hardware-based attacks. Furthermore, our concept aims at straightforward remote manageability for integration in IT infrastructures. This includes easy snapshot and recovery functionalities and, moreover, security updates independent of the smartphone manufacturer. The evaluation of our prototype implementation shows that our modifications introduce only a negligible performance overhead and reduce the battery lifetime by only 7.5 percent in the worst case.

This paper is organized as follows. In Section 2, we introduce our attacker model and in Section 3 an overview of virtualization techniques for Android is given, followed by a discussion of related work in Section 4. Section 5 introduces the basic concept of operating system-level virtualization for Android and our additional security mechanisms are described in Section 6. General implementation aspects are presented in Section 7 and our prototype is described in Section 8. Finally, we evaluate our results in Section 9 and conclude in Section 10.

2 Attacker Model

In our attacker model, we assume an attacker using common attack vectors on Android-based smartphones. This especially includes eavesdropping and modification of remote communication, installation of malicious applications on the device, and exploiting (known) vulnerabilities to gain access to higher privilege levels, typically root access. Besides remote attackers, we also consider local attackers with physical access to the device. However, it is not possible to protect the data and software running on the application processor against attacks using JTAG or similar mechanisms without modifications to the smartphone hardware, so such attacks are out of scope for this paper.

3 Virtualization Techniques for Android

Isolation mechanisms or rather virtualization techniques for Android can be classified in three groups, namely user-level isolation, operating system-level virtualization and system virtualization. Figure 1 shows these three basic concepts

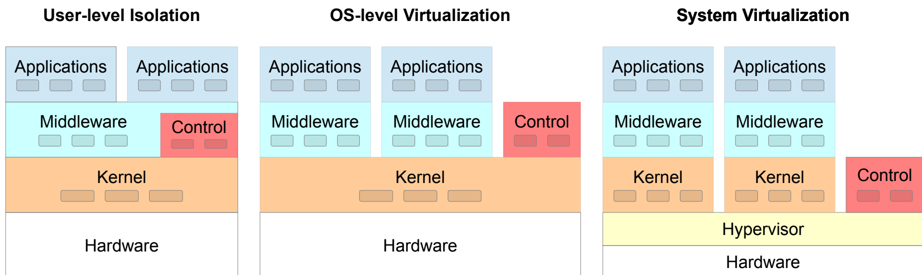

Fig. 1. Isolation and Virtualization Mechanisms for Android on three different Layers

for an example system with two isolated groups of applications and an additional control and management entity. By default, an Android system consists of an application layer, a middleware layer, and the kernel layer on top of the hardware. As shown in the figure, the main difference between the three concepts is which layers are shared by the isolated environments. Our concept is based on the architecture shown in the middle and is described in Section 5.

4 Related Work

The default Android security architecture uses different user identifiers (UIDs) per application group to implement a sandboxing mechanism. The communication between applications and core Android components is restricted based on permissions, which are requested during the installation of applications. It was shown that these mechanisms do not meet all security requirements [3,5], which led to a number of extensions to the Android architecture [4,11,10]. In contrast to our approach, these user-level isolation mechanisms usually require massive modifications to Android userspace components and introduce more complexity to the overall system.

Isolation based on operating system-level virtualization, as used in our approach, is a common concept of Unix-like operating systems today, especially on servers. In [2] *Cells* is introduced, a virtual mobile smartphone architecture, which utilizes Linux containers for isolation of two Android userspace instances running on one smartphone. In contrast to our approach, Cells does not focus on security. Specifically, it does not utilize access control policies and does not provide integrity protection or transparently encrypted and tunneled network connections.

Another approach to isolate runtime environments with different security requirements is system virtualization. System virtualization allows one to run multiple operating systems on one physical device using an additional software layer (a hypervisor [8] or microkernel [9]) as shown on the right side in Figure 1. This approach is also often used to add isolated security extensions to desktop or server systems [1,7,6]. Since current smartphone hardware does not provide hardware-assisted virtualization extensions, todays implementations usually require a paravirtualized kernel, like L4Android [9] for example. The main disadvantage of this approach is the complex and time-consuming act of porting software to new hardware and new Linux kernel versions. Furthermore, this approach usually results in a higher performance overhead when compared to operating system-level virtualization. This is mainly caused by additional context switches.

A trusted execution environment (TEE) is an isolated runtime environment for applications with high security requirements. These are typically used to implement a SE-like functionality on the same hardware as the normal system. Since access to TEEs is usually restricted by the device manufacturer and TEEs cannot provide the level of security that a SE can, we have chosen to use an external SE built into a microSD card for our prototype.

5 Lightweight Isolation Mechanism

Our concept is based on operating system-level virtualization, which provides userspace containers to isolate and control the resources of single applications or groups of applications running on top of one kernel as shown in the middle of Figure 1. This typically includes a unique hostname, process identifiers (PIDs), inter-process communications (IPCs), a filesystem, and network resources.

A trustworthy control and management environment is the first to run after boot and is the only component with full system access. Depending on the desired level of security, several Android userland instances are started from this environment or, alternatively, it ensures that only one additional Android userland instance is running at the same time. This means that processes are either frozen and not scheduled or stopped and removed from memory. In the following figures, we only depict a simplified system with one Android userspace instance.

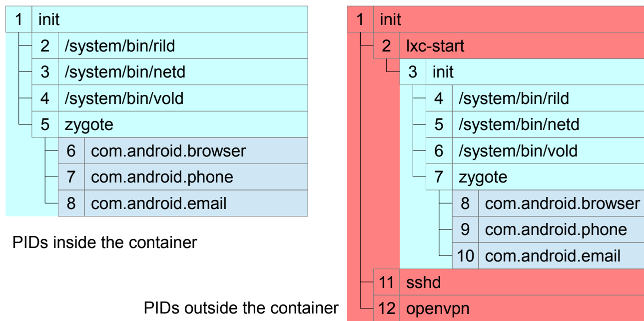

Fig. 2. Mapping of PIDs (Background Colors as in Figure 1)

Figure 2 shows the basic concept of operating system-level virtualization exemplarily for PIDs. From inside the container, only the processes corresponding to the particular container are visible. For an Android userland this looks like a process tree on a regular device. On the right side, the full system is shown including the processes 1, 2, 11 and 12 running outside of the container. The PIDs of the processes inside the container are mapped accordingly. Other resources are handled in a similar way.

A major advantage of this virtualization technique compared to system virtualization is that the isolation layer – here the kernel – has full control over all resources and can directly interfere at all processing layers and in all subsystems. This allows fine grained policy enforcement for system calls and integrity measurements of sensitive application groups at runtime. Additionally, for any input/output operation to devices, system call hooks can be used to add security extensions. We show four common implementation strategies of this approach in Section 7.

6 Integration of Security Mechanisms

In this section, we give an overview of security mechanisms and their integration in our basic concept. We systematically cover the three security aspects: isolation, communication, and storage. Most mechanisms are optional and can be applied to one or more containers if desired. The mechanisms are grouped into the following categories: remote management, capabilities and access control, network, storage, and display and user input.

6.1 Remote Management

A core component of our concept is a powerful remote management component. The trusted control and management component on the smartphone is isolated from the Android userspace through operating system-level virtualization as described in Section 5. It establishes a secure connection to a management server to fetch information. Alternatively, the management server can initiate the connection if the device has a public IP address or a VPN connection is already established (see Section 6.3). It is also possible to send an encrypted SMS to control the device. Since it is possible to send new binaries and scripts to the device, nearly everything can be triggered remotely. This also includes software updates independent from the device manufacturer to fix disclosed vulnerabilities and updates for integrity reference values and access control policies.

6.2 Capabilities and Access Control

On a stock Android system, the root user has full control over the system. Capabilities enhance the system security by enabling more fine-grained access control. This includes rebooting the system, configuring the network, loading kernel modules, and overriding file access permissions for example. Our concept ensures that all capabilities that are not necessary for an application to work properly are dropped systematically.

A similar approach is realized for access to devices, e.g., framebuffer, camera, network, and storage devices. So it is (remotely) configurable whether and when an Android container gets access to these devices.

To provide even more fine-grained access control, our concept provides access control for system calls based on well-known security models like mandatory access control (MAC) and access control lists (ACLs) as already utilized in other papers [11,13]. In our system, policies are typically configured by the administrator or automatically generated in learning mode or permissive mode. Another approach here is the automatic generation of policies on one reference smartphone, then slightly modified by an administrator, and finally the distribution to all managed devices.

6.3 Network and Telephony

A core component of our concept is the restriction of network and telephony services. Network filtering and routing can only be configured from the trustworthy

environment. This includes the routing of all connections from and to an Android userland through an encrypted virtual private network (VPN) tunnel as shown in Figure 3. In this scenario, the Android userland accesses a virtual ethernet interface (veth0), which is bridged (br0) with a tunnel interface (tun0) of a VPN. The asymmetric keys are stored in the SE and the negotiation of a symmetric session key is handled entirely in the SE. The secret key never leaves the SE. Additionally, to prevent unauthorized access to the VPN, the SE is protected with a personal identification number (PIN) as described in Section 6.5. Besides the network routing, in Figure 3 a scenario is shown, in which the Android userland cannot configure or access the WLAN interface (wlan0). This is only possible from the trustworthy part of the system. The authentication for encrypted WLANs can be handled similar to the VPN authentication.

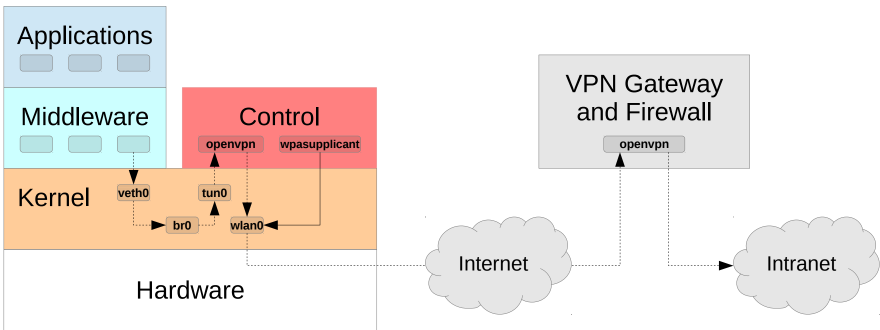

Fig. 3. Network with Transparent VPN

The second main communication channel to the outside world is the GSM/UMTS/LTE radio interface. Besides packet-based communication as described above, a common use case here are filter functions for SMS and calls. It is also possible to have separate phone numbers and connections for more than one Android container using VoIP for telephony as described in [2].

6.4 Storage

Virtualizing the root filesystem of the Android containers allows easy integration of snapshot and recovery functions and remote wipe functions as well as full and transparent root filesystem encryption. Furthermore, we apply integrity protection mechanisms to files as described in the following. Figure 4 shows a typical scenario with filesystem encryption and integrity protection controlled from the trustworthy control and management environment.

Encryption. Storage encryption can be integrated at the device level for a whole filesystem or at a per-file basis. Our default configuration uses an encrypted file system image, either in a single file or in a separate partition which

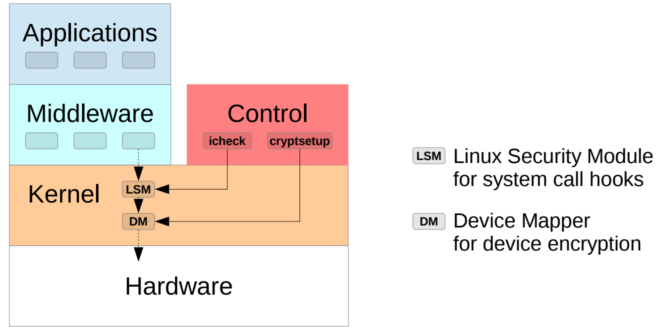

Fig. 4. Storage with Encryption and Integrity Protection

is mounted during the start of a container. The block-based symmetric encryption and decryption (e.g., aes-cbc-essiv) is handled by the kernel, which means that the key needs to be available in the kernel. Of course, the key is only accessible from the trustworthy environment during runtime and it is not accessible from Android containers. Furthermore, the key is never written unencrypted to persistent storage readable for an attacker on a switched off device. Instead, it is stored in the SE and protected with a PIN (see Section 6.5). Furthermore, if the security policy for the device allows only one active container at a time, the key is erased after the container is stopped and before another container is started. Highly sensitive information can also be encrypted and decrypted directly in a SE with a lower data throughput. Another option would be the utilization of a TEE-based implementation.

Integrity Protection. Basically, integrity protection can be enforced at the start of a container either on a whole filesystem or on a per file basis. This is especially useful after a recovery of the filesystem. Furthermore, our concept provides integrity protection of files while an Android container is running. Based on a whitelist or blacklist of file hashes, reading and/or writing to/from files is allowed or forbidden. This mechanism allows us to control which applications and system components are installed in the Android userland and to enforce that these applications and components fulfill certain requirements, like blacklisting of vulnerable versions of applications. It also allows blacklisting of known Malware. Our implementation for this mechanism is described in detail in Section 8. A similar approach could also be used for anomaly detection at runtime.

6.5 Display and User Input

A common problem for business and payment scenarios is the need for a trusted graphical user interface (GUI) for a secure password entry, e.g., to unlock a SE. Our concept uses a SE to store keys for several use cases and therefore also requires a secure password input dialog. Moreover, in our prototype switching from one Android container to another typically also requires a password.

To not introduce a performance impact, we usually give exclusive and full access rights to one Android container for the screen and touchscreen. To switch to the trustworthy environment, we use a hardware key (typically the power button). Additionally, we utilize the hardware LED of the device to indicate which environment receives the touchscreen events and which environment has access to the screen at the moment. Depending on the capabilities of the smartphone, different colors are associated to different containers (e.g., green for corporate, blue for private, red for guest and white for the trusted environment including the secure password entry). Of course, Android containers cannot directly access the LED driver any more.

7 Input and Output Hooks

In our concept, access to devices (like the LED) and other components needs to be restricted and sometimes virtualized to ensure the system's security. In this section, we give an overview of the generic approaches used in our implementation. Figure 5 shows four implementations to interfere with input and output operations to devices. Depicted are the data and control flow and the (un)modified components. The four implementation concepts are described in the following.

1. An unmodified Android userland component sends and receives data directly to/from a kernel component which has direct access to the hardware. The Android userspace has no permission to control the kernel component. This is reserved for the control and management environment. An example for this scenario is storage encryption, whereby the Android userland cannot set or get the encryption key (see Section 6.4).
2. An unmodified Android userland component sends and receives data directly to/from a kernel component which forwards this data to a trusted control component which itself forwards the data to a kernel component. This is typically used to implement a filter or access control mechanism in a trusted userspace component.

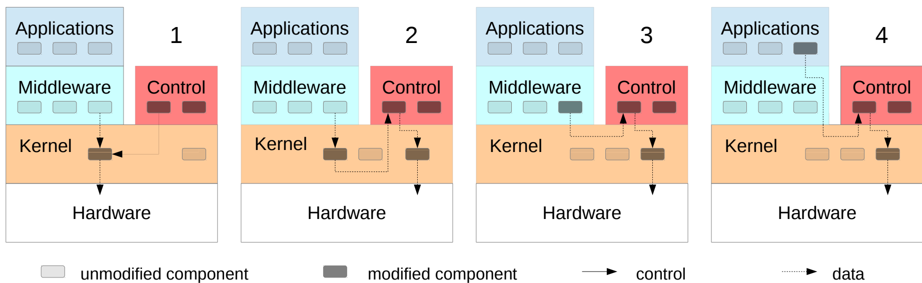

Fig. 5. Implementations to Interfere with Input and Output Operations of Containers

3. A paravirtualized Android userland component sends and receives data to/from a trusted userland component. The communication can be based on a shared memory segment or another IPC mechanism. For filtering purposes, this approach can provide already preprocessed data, which might be simpler to handle. An example for this scenario is the replacement of the rild or wpa_supplicant binaries in the Android userspace with stubs, if the Android userland does not have the permission to directly configure the radio modem and WLAN interface. On a device with more than one Android userland, this approach can also be used to share components between the Android userlands, e.g., the address book, or for multiplexing devices.
4. A special Android application communicates with a trusted component. This scenario can be used to allow an Android application to call a trusted dialog implemented in a trusted component outside the Android container, e.g., a password entry dialog. This can be useful for a modified Email application with SE-based S/MIME signatures.

8 Prototype Implementation

Our prototype implementation uses Debian GNU/Linux for the trusted control and management environment and to verify the portability of our approach, we used a rather old Android 2.3.5 on a *Google Nexus One* and a recent Android 4.0.4 on a *Samsung Galaxy S3* for the Android userspace instances. In both cases, the filesystems are stored on a microSD card. The stock kernel was modified to support Linux containers for operating system-level virtualization. This mainly includes resource isolation based on namespaces and resource control based on Linux kernel control groups (cgroups). Most capabilities are dropped for containers and access to devices etc. is restricted. Most of our additional concepts are implemented based on the generic approaches described in Section 7. However, one primary aspect of our approach is described in the following in more detail.

Our access control and integrity protection mechanisms are based on Linux Security Modules (LSM) [12], which provide lightweight, general support for access control by allowing modules to define security hooks for system calls. This allows a straightforward integration of task hooks, program loading hooks, IPC hooks, filesystem hooks, network hooks, module hooks (e.g., module initialization) and system hooks (e.g., hostname setting). The called hook function can allow or deny the requested access. This approach has also been used by other research papers, which utilize Linux Security Modules on an Android system without operating system-level virtualization. These typically utilize a module directly included in the mainline Linux kernel, like TOMOYO or SELinux [11,13]. Of course, without a container-based virtualization environment, all userspace components need to be included in the Android userspace. Here, we have a significant advantage in our virtualized system where the userspace components run in the trustworthy environment.

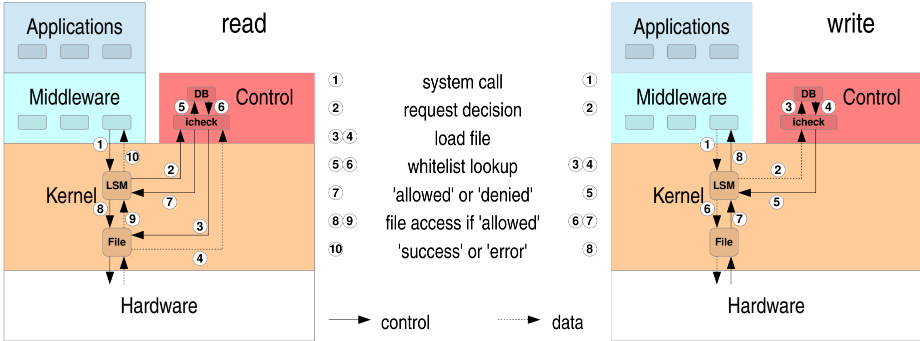

Fig. 6. Data and Control Flow for File-based Integrity Measurements

As an example, in Figure 4 a typical storage scenario with two security extensions is shown. First, we use the device mapper (DM) in the kernel for transparent encryption. It is controlled by the userspace tool *cryptsetup* running in the trustworthy environment. Second, for integrity protection, we use a LSM implementation controlled by our userspace daemon *icheck*. Figure 6 shows the control and data flow for file-based read (left side of the figure) and write (right side of the figure) operations. In our prototype implementation, this daemon calculates a SHA1 of files on file accesses and compares this hash with a whitelist of hashes stored in the trustworthy environment. If a hash is not found in the whitelist, access will be denied. The whitelist for system components is usually distributed with the system image and the whitelist of Android applications can either be distributed via the remote management interface or directly generated on the device by verifying and analyzing Android application packages (apk) including its cryptographic signatures.

9 Evaluation and Measurements

In this section, we present measurements regarding the performance impact, the power consumption, and the memory usage of our prototype implementation. The measurements show that our approach has a very limited impact on the performance of the system and especially the power consumption. Finally, a security evaluation is given.

All measurements were done for three Android environments running on a Google Nexus One with equal configuration parameters and with the same set of Android system services and applications running in the background:

1. **Default Android.** An Android userland is running directly on the system without operating system-level virtualization and without a trusted control and management entity. This is the reference value in the following figures (7, 8 and 9).

2. **Container.** An Android userland is running inside a Linux container. All security extensions which may result in an overhead are disabled.
3. **Encrypted Container.** An Android userland is running inside a Linux container and the whole filesystem of this container is encrypted.

9.1 Power Consumption

Power consumption of the device is measured in three typical usage scenarios. In the first scenario, the device runs continuously in the idle state without communication over WLAN or cellular and with display backlight turned off. In the second scenario, a music player application runs in the foreground while the display is still turned off. In the third scenario, the power consumption is measured during the usage of the web browser. Here, the display is turned on and user interaction is simulated every five minutes. Figure 7 shows our results.

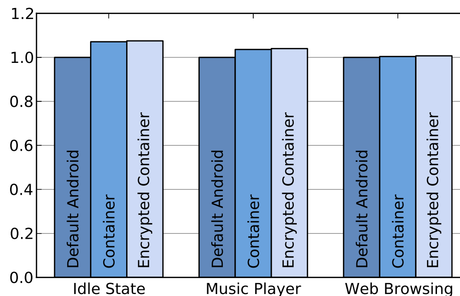

Fig. 7. Normalized Battery Overhead for Default Android and Containers

The highest impact on the power consumption was measured in the idle test scenario with 7.5 percent for the encrypted container and 7.1 percent for an Android container without storage encryption. For the music player scenario, the impact goes down to 4.0 and 3.6 percent and for the web browsing scenario it's below 1 percent for both containers. This increased power consumption is mainly a result of additional threads running outside the Android container in the trustworthy environment.

9.2 Performance

We run six Android benchmarks to measure the performance impact of our approach compared to a default Android userland without operating system-level virtualization. The results are shown in Figure 8.

Most benchmarks show less than 1 percent variation in performance overhead for Android containers. This shows that our approach provides nearly native performance for an Android userland running inside a Linux container. However, there are some values, which will be explained a little more closely.

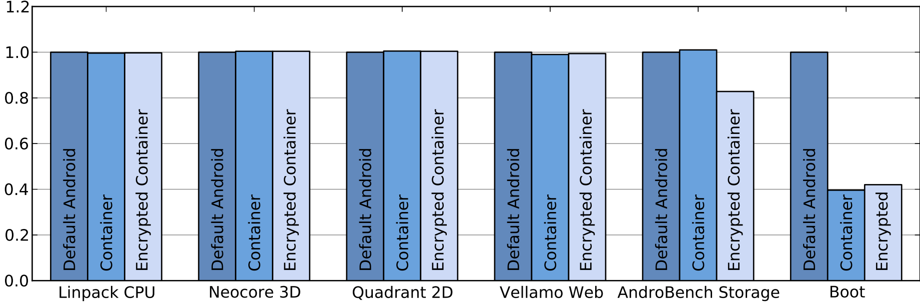

Fig. 8. Normalized Runtime Overhead for Default Android and Containers

The *AndroBench Storage* benchmark shows a higher performance for an Android container with encrypted root filesystem. This can be explained by an additional caching of the encrypted filesystem image, which cannot be prevented by the benchmark tool. Next, the boot procedure of an Android container is considerably faster due to different caching behavior, as shown in the next section.

9.3 Memory Usage

In Figure 9, the memory usage of the whole system is shown. A system with operating system-level virtualization naturally results in a slightly higher memory usage (depicted red). Noticeable here is the much higher amount of memory used for caching (depicted blue). For our measurements, we considered two system states. First, the memory usage of the system after the boot (depicted on the left) and second, the increase of memory usage after the start of additional applications (depicted on the right).

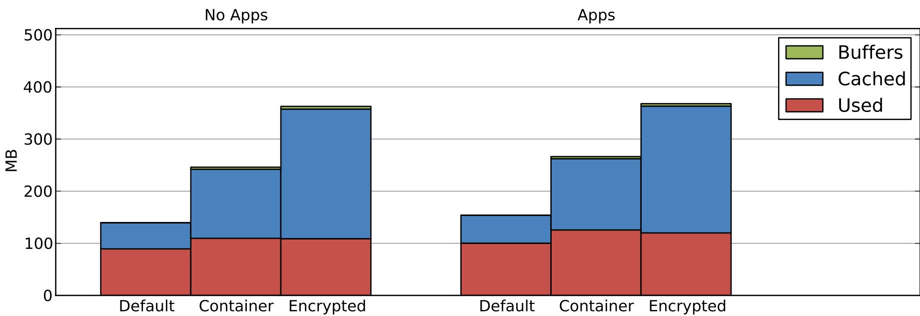

Fig. 9. Memory Usage for Default Android and Containers

9.4 Security Evaluation

Our implementation is based mainly on two Linux kernel-level security mechanisms, namely Linux containers including namespaces and control groups, and Linux security modules. Because of our systematic approach to drop as many capabilities and privileges as possible and to restrict inter process communication, we can provide a higher security level than a stock Android system. In particular, getting root access rights to get full control over the system is a common attack vector on stock Android systems. In our system, root access in an Android container does not provide full control over the system and more importantly no access to confidential keys stored in the SE. Furthermore, practical tests showed that our integrity protection mechanism can reliably prevent the installation of unknown and therefore possibly malicious applications. Finally, we prevented real attacks using exploits for the `/dev/exynos-mem` device security hole found in the stock Samsung Galaxy S3 firmware (CVE-2012-6422).

Comparing our approach to other virtualization techniques for Android as described in Section 3 indicates the following. On the one hand, our implementation has a smaller trusted computing base (TCB) compared to user-level isolation. Not sharing the middleware layer simply means that attacks on this layer, as described in [5], are not possible between Android containers. On the other hand, system virtualization with a focus on security (e.g., based on a microkernel) has a smaller TCB. However, this approach typically has practical disadvantages as already mentioned, but can be combined with our approach, e.g., in form of a TEE. To protect highly sensitive information in our prototype, we utilize a hardware SE to provide an even higher level of security including hardware-based attacks.

10 Conclusion

In this paper, we presented a lightweight mechanism to isolate one or more Android userland instances from a trustworthy control and management environment. In contrast to existing solutions based on full system virtualization, our approach requires no complex software modifications.

Additionally, we implemented several security extensions. A key functionality is the easy remote administration and management of mobile devices. Further key features are transparent encryption and tunneling of network connections and transparent storage encryption, where the Android userland does not have access to the used cryptographic keys. Moreover, we implemented integrity protection mechanisms and a secure GUI for password entry dialogs etc.

Our evaluation results show that our approach is practical and introduces only a negligible performance overhead and reduces the battery lifetime by only 7.5 percent in the worst case.

Acknowledgments. Parts of this work were supported by the German Federal Ministry of Education and Research (BMBF) under grant 01BY1011 within the project ASMONIA.

References

1. Alkassar, A., Scheibel, M., Stübel, M., Sadeghi, A.R., Winandy, M.: Security Architecture for Device Encryption and VPN. In: ISSE 2006 – Securing Electronic Busines Processes, pp. 54–63. Vieweg (2006)
2. Andrus, J., Dall, C., Hof, A.V., Laadan, O., Nieh, J.: Cells: A Virtual Mobile Smartphone Architecture. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 173–187. ACM, New York (2011)
3. Barrera, D., Kayacik, H.G., van Oorschot, P.C., Somayaji, A.: A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 73–84. ACM, New York (2010)
4. Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., Shastry, B.: Practical and Lightweight Domain Isolation on Android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2011, pp. 51–62. ACM, New York (2011)
5. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege Escalation Attacks on Android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
6. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A Virtual Machine-Based Platform for Trusted Computing. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 193–206. ACM, New York (2003)
7. Hartig, H., Hohmuth, M., Feske, N., Helmuth, C., Lackorzynski, A., Mehnert, F., Peter, M.: The Nizza Secure-System Architecture. In: International Conference on Collaborative Computing: Networking, Applications and Worksharing (2005)
8. Hwang, J.Y., Suh, S.B., Heo, S.K., Park, C.J., Ryu, J.M., Park, S.Y., Kim, C.R.: Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In: 5th IEEE Consumer Communications and Networking Conference, CCNC 2008, pp. 257–261 (2008)
9. Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., Peter, M.: L4Android: A Generic Operating System Framework for Secure Smartphones. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2011, pp. 39–50. ACM, New York (2011)
10. Ongtang, M., Butler, K., McDaniel, P.: Porscha: Policy Oriented Secure Content Handling in Android. In: Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC 2010, pp. 221–230. ACM, New York (2010)
11. Shabtai, A., Fledel, Y., Elovici, Y.: Securing Android-Powered Mobile Devices Using SELinux. IEEE Security and Privacy 8(3), 36–44 (2010)
12. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux Security Modules: General Security Support for the Linux Kernel. In: Proceedings of the 11th USENIX Security Symposium, pp. 17–31. USENIX Association, Berkeley (2002)
13. Zhang, X., Aciicmez, O., Seifert, J.P.: A Trusted Mobile Phone Reference Architecture via Secure Kernel. In: Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing, STC 2007, pp. 7–14. ACM, New York (2007)