A Practical Hardware-Assisted Approach to Customize Trusted Boot for Mobile Devices

Javier González[1], Michael Hölzl[2], Peter Riedl[2], Philippe Bonnet[1], and René Mayrhofer[2]

[1] IT University of Copenhagen, Denmark
{jgon, phbo}@itu.dk
[2] University of Applied Sciences Upper Austria, Campus Hagenberg
{michael.hoelzl, peter.riedl, rene.mayrhofer}@fh-hagenberg.at

Abstract. Current efforts to increase the security of the boot sequence for mobile devices fall into two main categories: (i) secure boot: where each stage in the boot sequence is evaluated, aborting the boot process if a non expected component attempts to be loaded; and (ii) trusted boot: where a log is maintained with the components that have been loaded in the boot process for later audit. The first approach is often criticized for locking down devices, thus reducing users' freedom to choose software. The second lacks the mechanisms to enforce any form of run-time verification. In this paper, we present the architecture for a two-phase boot verification that addresses these shortcomings. In the first phase, at boot-time the integrity of the bootloader and OS images are verified and logged; in the second phase, at run-time applications can check the boot traces and verify that the running software satisfies their security requirements. This is a first step towards supporting usage control primitives for running applications. Our approach relies on off-the-shelf secure hardware that is available in a multitude of mobile devices: ARM TrustZone as a Trusted Execution Environment, and Secure Element as a tamper-resistant unit.

Keywords: Secure Boot, Trusted Boot, Secure Element, TrustZone

1 Introduction

Today, mobile devices are designed to run a single Operating System (OS). Typically, Original Equipment Manufacturers (OEMs) lock their devices to a bootloader and OS that cannot be substituted without invalidating the device's warranty. This practice is supported by a wide range of service providers, such as telecommunication companies, on the grounds that untested software interacting with their systems represents a security threat[3]. In the few cases where the OEM allows users to modify the bootloader, the process is time consuming, requires a computer, and involves all user data being erased[4]. This leads to OEMs indirectly deciding on the functionalities reaching the mainstream. As a consequence, users

[3] http://news.cnet.com/8301-17938_105-57388555-1/verizon-officially-supports-locked-bootloaders/
[4] http://source.android.com/devices/tech/security/

seeking the freedom of running the software that satisfies their needs, tend to root their devices. However, bypassing the security of a device means that these users lose the capacity to certify the software running on it. This represents a risk for all parties and services interacting with such a device [17, 21].

This topic has been widely discussed by Cory Doctorow in his talk *Lockdown: The Coming Civil War over General Purpose Computing* [6]. Here, he argues that hardware security platforms such as Trusted Platform Module (TPM) have been misused to implement what he calls the lock-down mode: *"Your TPM comes with a set of signing keys it trusts, and unless your bootloader is signed by a TPM-trusted party, you can't run it. Moreover, since the bootloader determines which OS launches, you don't get to control the software in your machine."*. Far from being taken from one of his science fiction dystopias, the lock-down mode accurately describes the current situation in mobile devices: users cannot always modify the software that handles their sensitive information (e.g. pictures, mails, passwords), control the hardware peripherals embedded in their smart-phones (e.g. GPS, microphone, camera), or connect to their home networked devices (e.g. set-top boxes, appliances, smart meters). This raises obvious privacy concerns.

In the same talk, Doctorow discusses an alternative implementation for hardware security platforms - the certainty mode -, where users have the freedom to choose the software running in their devices, and the *certainty* that this software comes from a source they trust. What is more, he envisions the use of context-specific OSs, all based on the user's trust. The issue is that the trust that a user might have in a given OS, does not necessarily extended to the third parties interacting with it (e.g., private LANs, cloud-based services, etc.).

In this paper we present an approach to allow users choosing the OS they want to run in their mobile devices while (i) giving them the certainty that the OS of their choice is effectively the one being booted, and (ii) allowing running applications to verify the OS in run-time. Put differently, we extend Doctorow's certainty mode idea to all the parties interacting with a mobile device. While modern Unified Extensible Firmware Interface (UEFI) platform support the installation of new OSs and their corresponding signing key by means of the BIOS, this is not the case for mobile platforms, where TPM is not present. The same applies for user space applications doing integrity checks on the running system. In order to address this issue in mobile devices we propose a two-phase verification of the boot process: in the first phase, boot components are verified and logged in the same fashion as trusted boot; in the second phase, the boot trace can be checked by running applications in order to verify the running OS. We base the security of our design in hardware security extensions present on a wide range of mobile devices: ARM TrustZone as a Trusted Execution Environment (TEE), and Secure Element (SE) as a tamper-resistant unit.

2 Related Work

The architecture of a secure boot process for desktop PCs was first proposed by Arbaugh et al. in [3]. Their architecture, called AEGIS, describes a way to

verify the integrity of a system by constructing a chain of integrity checks. Every stage in the boot process has to verify the integrity of the next stage. After this first description of a secure bootstrap process, multiple specifications and implementations of such a system have been created. One of the first specifications that support this feature was proposed by the Trusted Computing Group (TCG) in conjunction with the Trusted Platform Module (TPM) standard [22]. A TPM is a secure cryptographic-coprocessor embedded in the PC architecture and provides a set of functionalities, such as generation of cryptographic key-pairs, a random number generator and protected storage. **Trusted boot** [7] is the implementation which makes use of this hardware module to verify the boot sequence. A machine running with trusted boot sends the hash of each following stage in the boot sequence to the TPM where it will be appended to the previous hash. This creates a hash chain, called Platform Configuration Register (PCR), that can be used for various purposes. For example, it can be used to decrypt data only when the machine reached a specific stage in the boot sequence (*sealing*) or to verify that the system is in a state that is trusted (*Remote Attestation*). The TCG Mobile Phone Working Group proposed a concept on how to implement such a trusted boot also on mobile devices using a TPM-like hardware component called Mobile Trusted Module (MTM) [19]. Another boot verification protocol is **secure boot**, described in the UEFI specifications since version 2.2 [23]. UEFI secure boot verifies the integrity of each stage by computing a hash and comparing the result with a cryptographic signature. A key database of trustworthy public keys needs to be accessible during boot time with which the signature can be verified. If verification fails, the system will abort the boot process. Due to this reason, and the fact that only a limited amount of keys are pre-installed on the platform, the implementation of this system has been criticized of preventing users to install an OS of their choice.

Although both systems, TPM trusted boot and UEFI secure boot, are widely spread on desktop computers, they still did not reach the mobile platform: the efforts to port UEFI to ARM devices - mainly driven by Linaro - have been publicly restricted to ARMv8 servers, not considering mobile or embedded platforms[5]. Also the MTM, though especially designed for mobile devices, has not been integrated into current device hardware (except for the Nokia N900 in 2009). This leads to the necessity for different solutions in current off-the-shelf mobile devices.

3 Background

3.1 Secure Hardware Support

The goal of the two-phase boot verification is to have a bootstrap architecture that can be trusted and easily customized by the owner of the mobile device. For our approach to be deployable only by means of a software update, it is necessary that it is based on off-the-shelf secure hardware components already deployed in mainstream mobile devices.

[5] http://www.linaro.org/blog/when-will-uefi-and-acpi-be-ready-on-arm/

Secure Element (SE). The SE is a special variant of a smart card, which is usually shipped as an embedded integrated circuit in mobile devices together with Near field Communication (NFC) [18] and is already integrated in a multitude of mobile devices (e.g., Samsung Galaxy S3, S4, Galaxy Nexus, HTC One X, etc.). Furthermore, a secure element can also be added to the device with a microSD or an Universal Integrated Circuit Card (UICC). The main features of a SE are: **data protection** against unauthorized access and tampering, **execution of program code** in form of small applications (applets) directly on the chip and the **hardware supported execution of cryptographic-operations** (e.g., RSA, AES, SHA, etc.) for encryption, decryption and hashing of data without significant run-time overhead [12].

Trusted Execution Environment (TEE). A TEE is a secure environment inside a computing device that ensures that sensitive data is only stored, processed and protected by authorized software. The secure environment is separated by hardware from the device's area running the main OS and user applications, which is denoted rich environment.

An example of TEE is ARM TrustZone [4]. TrustZone relies on the so-called NS bit, an extension of the AMBA3 AXI Advanced Peripheral Bus (APB), to separate rich and secure environments. The NS bit distinguishes those instructions stemming from the secure environment and those stemming from the rich environment. Access to the NS bit is protected by a gatekeeper mechanism referred to as the secure monitor, which is triggered by the System Monitor Call (SMC). The OS thus distinguishes between user space, kernel space and secure space, where only authorized software runs in secure space, without interference from user or kernel space. Also, any peripheral connected to the APB (e.g., interrupt controllers, timers, and user I/O devices) can be configured by means of the TrustZone Protection Controller (TZPC) virtual peripheral to have prioritized - or even exclusive - access from the secure environment. Since TrustZone enabled processors boot always in secure mode, secure code executes before the general purpose bootloader booting the rich OS (e.g., u-boot) is even loaded in memory. This allows to define a security perimeter formed by code, memory and peripherals, making TrustZone a good candidate to support trusted boot solutions as the one presented in this paper.

While TrustZone was introduced 10 years ago; it is first recently that Trustonic, Xilinx and others have proposed hardware platforms and programming frameworks that makes it possible for the research community [8] as well as the industry to experiment and develop innovative solutions with TrustZone.

3.2 Threat Model

The main motivating goal of our approach for a two-phase boot verification is to give the user the certainty that the OS of their choice is indeed the one being booted and that a malicious entity is not able to get access to sensitive data. Hence, our threat model on mobile devices considers several kind of software and hardware attacks:

Software Attacks. Our threat model for software attacks only concerns attacks within the software stack of the mobile device. This includes attacks on application level, OS level and down to kernel/driver level.

With the increase of mobile devices and amount of security sensitive applications running on them, we can expect a growing number of mobile malware **exploiting errors of the OS** [15]. Attacks might be carried out to read sensitive data within standard application permissions or - in the worst case -, exploit privilege escalation [11, 20]. Certain types of mobile malware are able to directly infect the OS to achieve their malicious goal (i.e. *rootkits*). **Communication** between applications, libraries, kernel drivers, etc. is also subject to different kinds of attacks. Mobile malware such as trojans or viruses running on the device might have the ability to interfere with the data path and gain full control over exchanged messages (e.g., service hijacking [5], library call interception [16]). This is specially relevant when the communication affects hardware components handling sensitive information (i.e., secure data path). An adversary seeking to compromise the secure data path can be assumed to be able to perform various types of attacks: eavesdropping, data injection, denial-of-service, man-in-the-middle, etc.

Hardware Attacks. As mobile devices become ubiquitous, the chances of them being lost or stolen have increased substantially. This results in physical attacks being increasingly relevant. Malicious hardware possession enables threats through both physical tampering of the hardware and software modification of the device. An attacker can take advantage of having physical access to the device to either load a malicious OS or bootloader in order to bypass the hardware protection, or directly attempt to access sensitive information (i.e., keys and secrets) from secondary storage (e.g., flash memory). Even if the file system is encrypted using tools provided by the OS, it is possible for an attacker with physical access to the device to circumvent them [10]. Protection against these attacks using hardware based solutions assuring tamper resistance have been proposed in order to increase the protection of sensitive data on mobile devices [13].

4 Example Scenario

The solution we propose in this paper enables providers of services or applications with security concerns to adapt an OS to meet their particular security requirements. This customized OS is offered by a trusted issuer; therefore we call it a **certified OS**. Creating a certified OS could mean to restrict the installation of applications, restrict network access, restrict access to hardware, etc. By allowing users to exchange OSs that can be certified, we enable services and organizations to establish restrictions concerning the software that interacts with their systems, while preserving the user's right to choose - and certify - the software handling their personal information.

Bring Your Own Device (BYOD). One practical application for a certified OS could be the BYOD problem. This refers to users wanting to interact from

their personal mobile devices with their company IT services. Recent studies show that the BYOD problem is growing [1]. Given the heterogeneity of the current mobile device OS landscape, this can create overhead for system administrators and impede productivity. The multitude of different versions of each OS combined with adaptions to the OS by hardware manufacturers create this fragmented OS landscape. Concrete challenges include porting services to different platforms, having to deal with platform-specific security threats, or increasing the complexity of the enterprise LAN since the devices connecting to it cannot be trusted. By using a customized OS that is preconfigured to interact with all company services, enterprises could save time and money while increasing the security of their services. The effort to adapt OSs can be reduced by supporting only one or few versions of the most common OS (Android and iOS combined currently have a market share of 96%[6]). When employees are not using company services, they can switch to an OS they trust to handle their personal information. In this way one single device can be used both privately and professionally with the minor trade-off of having to switch OS.

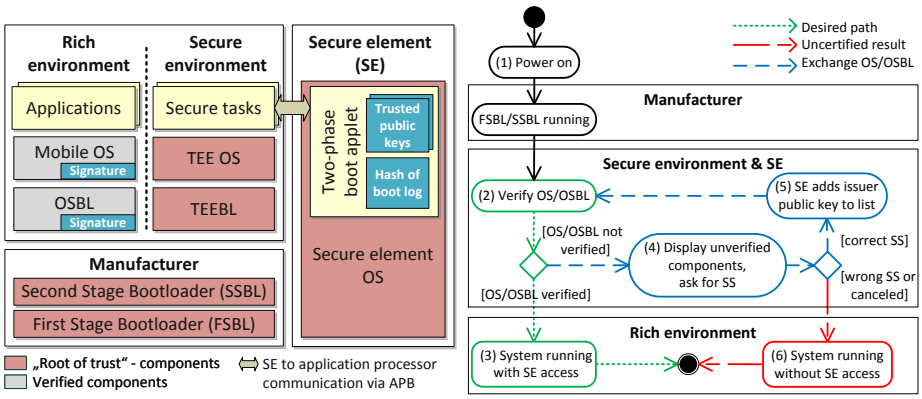
The aforementioned BYOD applications serve as good examples for sensitive data. All credentials needed to authenticate to enterprise services like email and VPN private keys could be stored in the tamper resistant SE (see Section 3.1). Access to that data would only be granted if the request comes from a certified OS and all contextual requirements are met (e.g., being inside of the enterprise's VPN).

5 Architecture

Figure 1a depicts how hardware and software components interact with each other in the two-phase boot verification. In normal operation, both, the rich and the secure environment are booted. This is, the signatures of the mobile OS and the OS bootloader (OSBL) have been validated by the SE. Note that the verification process is carried out by the trusted execution environment (TEE), consisting of its bootloader (TEEBL) and an OS (TEE OS), and the SE as root of trust. Additionally, we assume the manufacturer bootloaders, first (FSBL) and second stage (SSBL), also as root of trust components. Applications running in the rich area can be installed and executed without restrictions just as we are used to see in current mobile OSs. We do not make any assumptions regarding the trustworthiness of these applications. We assume however that the secure tasks running in the TEE can be trusted. We describe the communication between the rich and the trusted execution environment in more detail in Section 3.1.

The SE is configured as a trusted peripheral and therefore only accessible from secure tasks in the TEE. Note that while the SE is not a peripheral in itself, it is connected via the APB (e.g., I2C, etc.), and therefore treated as such. In this way, rich applications make use of secure tasks to access the SE. As a consequence, if the TEE becomes unavailable, rich applications would be unable

[6] http://www.idc.com/prodserv/smartphone-os-market-share.jsp



(a) (b)

Fig. 1: Components of our proposed customizable two-phase boot verification for off-the-shelf mobile devices in (a). The activity diagram in (b) describes the first phase. Transitions show the desired path (dotted), an uncertified result (long-dashed) and steps used for exchanging OS (dashed). "SS" stands for shared secret.

to communicate with the SE. The TEE becoming unavailable could be a product of a failure, but also a defense mechanism against unverified software. If the OS image cannot be verified, the OS would still be booted, however the TEE will not respond to any attempt of communication. In this case we give up on availability in order to guarantee the confidentiality and integrity of the sensitive data stored in the SE (e.g., corporate VPN keys stored in the tamper-resistant hardware). Finally, the TEE also serves as a medium between certificates of the OS issuers in an untrusted storage (e.g., SD card, flash memory) and the SE. This gives users the option to add additional trusted public keys to the SE when the first-phase verification failed (see Section 5.3 for details).

5.1 First phase verification

The first phase verification is responsible for verifying and starting the TEE and the OS, and for enabling the user to install a customized OS and flag it as trusted. Figure 1b depicts the process (numbers in the diagram correlate with the enumeration, borders indicate which component is responsible for each step).

1. The user presses the power button. FSBL, SSBL, and TEE are started.
2. The TEE attempts to verify OS/OSBL by sending their signatures and the hash of the images to the SE. The SE attempts to verify the signatures with all trusted public keys of OS/OSBL issuers and compares it to the received hash. The result of this operation is reported to the TEE. In addition, the hash of the booted OS for the second-phase is stored in the SE. Several hashes using different algorithms are calculated to better support phase two (Section 5.2). Since the images can be cached, the transmission cost is only paid once.

3. Once the OS/OSBL are verified, the OS is booted with full access to the SE through the TEE. This is the end of the expected course of actions.
4. If the verification of OS/OSBL (step 2) fails, a message, explaining which component is unverified, is displayed. Now the user can choose to either certify the OS/OSBL by entering the shared secret and provide an issuer certificate, or continue to boot the uncertified OS/OSBL. For further details see Section 5.3.
5. If a legitimate user (authenticated by shared secret) flags an OS/OSBL as trusted, the SE adds the public key of the issuers' certificate to the list of trusted keys and continues to verify the newly added OS.
6. If the user enters a wrong shared secret or cancels the operation, the uncertified OS/OSBL is booted without access to the SE and a message informs the user about the uncertified state. This is the end of the uncertified course of actions, where the user can still use the device (uncertified OS/OSBL is booted), but due to missing verification, access to the secured data is denied by the TEE.

This architecture ensures that only a combination of verified OS and OSBL are granted access to the SE. If one of the components is not verifiable, the device can still be used (uncertified OS is booted), but access to the SE is denied. In the first phase all executed commands are logged in the SE by maintaining a hash chain. This approach is similar to trusted boot, and enables the second phase verification. The process of installing custom OS is explained in Section 5.3.

5.2 Second phase verification

In the second phase verification, rich applications can verify the system in which they are running before executing. To do this, rich applications make use of a secure task in the TEE to validate the running OS by checking the boot traces in the SE. This secure task is not application-specific, but a *secure* system primitive that any rich application running in user space can request to use.

In ARMv7, the SMC call needs to be issued from a privilege mode; this means that either the user application executes a Supervisor Call instruction (SCV) to enter in supervisor mode, or the SMC call stems from a place already executing in privilege mode. Relying on user applications to call the monitor and handle the communication with a secure task that is shared among different user applications can compromise the verification coming from the secure environment (e.g., by compromising application binaries). Another alternative would be letting applications handling the boot trace verification by having each of them implementing their own secure task. However, this solution would not only result in code replication, but in an unnecessary increase of the Trusted Computing Base (TCB), and therefore in the secure environment having a larger attack surface. Also, any update concerning the verification of the boot trace would need to propagate to many different secure tasks, therefore slowing down the dissemination of critical security patches[7]. By defining the system verification as

[7] Cases like the Heartbleed Bug (http://heartbleed.com) are good examples of how a rapid dissemination of a security patch is necessary.

a system primitive, the kernel makes the process transparent to user applications, addressing all these issues. This approach does not assume the kernel to be trusted; on the other hand, it limits the attack surface to the communication channel (the kernel), avoiding sensitive code to be delegated to user space. In Section 6 we address this is detail as part of the security analysis.

In this way, rich applications can communicate with the kernel, who forwards the call to the secure task that verifies the OS image. Rich applications pass the list of hashes they trust and a parameter defining the hash algorithm used to calculate them (e.g., SHA, DJB2, MD5). This allows for several hashing algorithms being supported and new ones being easily introduced. When the secure task executes, it request the OS image hash to the SE specifying the algorithm, and compares it with the hashes trusted by the rich application calling it. If the hash using the required algorithm is not precomputed, the TEE requests a reboot and adds the algorithm to the SE. Applications can also verify the boot traces to check that all the components they require (e.g., biometric sensor) have been correctly initialized. As a result, rich applications are able to make decisions in run-time depending on both the OS and the available peripherals. This can be seen as a first step towards supporting usage control.

Since the SE is configured as a trusted peripheral, rich applications cannot directly communicate with it; they need to do it through the TEE. Additionally, the SE signs the retrieved hashes using its own private key in a similar manner as the TPM using the Attestation Identity Key (AIK). To distribute the corresponding public keys of the SE, an infrastructure similar to other public-key infrastructures is required (e.g., openPGP).

5.3 Exchange OS/OSBL

We share the concern that current secure boot implementations necessarily lock devices to a specific OS chosen by the OEM. In order to avoid this in our architecture, we propose the **configuration mode**. TEEs are suitable candidates to implement this mode, since they support features for secure user interactions (see Section 3.1). The sequence of actions for the configuration mode starts by the user flashing a new mobile OS or OS bootloader (e.g., using uboot) with a signature that is not trusted by the platform (public key of the OS issuer is not in the list of trusted keys). As depicted in the transition from step 2 to 4 in Figure 1b, the OS will not be booted in that case. The user will now be given the possibility to either manually verify the signature of the new mobile OS or cancel the process within a secure task of the TEE. In case of a requested manual verification, the user will be asked to point to the certificate of the OS issuer on the untrusted memory (e.g., SD card) and enter a shared secret within the secure UI of the TEE (step 4 in Figure 1b). This shared secret could be a PIN or password that has been shipped together with the SE. With an appropriate secure channel protocol (e.g., SRP [24]), the user will be authenticated to the SE and a secure communication between TEE and the applet will be established. If the user does not want to verify the OS, the system would still be booted without access to the sensitive data in the SE (step 6 in Figure 1b). After successful

authorization, the secure task sends the public key of the OS issuer to the applet, where it will then be added to the list of trusted keys (step 5 in Figure 1b). If users do not have access to the certificate, or do not want to completely trust the issuer, they can also exclusively sign the specific OS instance with the private/public key pair of the SE. Attacks attempting to flag an OS as trusted will fail as long as the shared secret remains unknown to the attacker. An adversary could also try to manipulate the certificate which is stored in the untrusted memory. However, as the TEE has full network capabilities, it can verify the certificate validity with a correspondent public-key infrastructure, such as the web of trust from PGP.

6 Security Analysis

There are four kinds of attacks that can be perpetrated against the two-phase verification architecture design: (i) attacks against the manufacturer bootloaders to prevent verifying and logging the loaded components during the boot process; attacks against the TEE rich - secure interface through (ii) attacks against the secure monitor, and (iii) attacks against the secure data path; and (iv) physical attacks against the SE to steal the secrets stored in it. Most of the OS specific services we describe either as exploits or defenses assume a Linux based OS. While not all of these services are available in all OSs, they are conceptually independent to a specific implementation.

Bootloaders. Since the root of trust begins with the FSBL and SSBL, a sophisticated software attack that supplants the SSBL could prevent the boot of a legitimate TEE, and therefore prevent the verification and logging of booted components depicted in Figure 1b. While the attacker would gain control of the device and the communication to the SE, the secrets stored in the SE would remain inaccessible at first. Indeed, the SE applet is configured to wait for a trusted system state, thus it will not reveal sensitive information if the correspondent boot hashes are sent. The attacker would need to modify the SSBL so that it is sending the hashes of a normal trusted boot. As the SE is only a passive component, it does not have methods to verify the trustworthiness of the source of a received message. Signing the messages would also not prevent these attacks due to the inability to securely store the private key on the mobile device. While intricate, the attack is theoretically possible. However, we assume that the SSBL is locked by the OEM and additionally verified by the FSBL, as it is the case in current devices. Still, the lack of source verification capability of the SE applet remains an open challenge for the research community. If an attacker only substitutes the OS bootloader, an untrusted OS would be booted without access to the SE. This is already one of the scenarios contemplated as normal operation (i.e. step 6 in Figure 1b).

Secure Monitor. The secure monitor is in charge of switching between the rich and secure environments, thus representing TrustZone's most vulnerable component. If compromised, illegitimate applications could run while the processor

is executing in secure mode. This involves prioritized access to all peripherals and memory. The fact that the SMC call for ARMVv7 architectures is implemented in software together with the lack of a standard implementation has led to bad designs like the one reported in October 2013, affecting Motorola devices running Android 4.1.2. (The attack reached the National Vulnerability Database and scored an impact of 10.0 (out of 10.0)[8], and it can be found in the attacker's blog[9]). For ARMv8, however, ARM is providing an open source reference implementation of secure world software called ARM Trusted Firmware. This effort includes the use of Exception Level 3 (EL3) software [9] and a SMC Calling Convention. Organizations such as Linaro are already pushing for its adoption.

Secure Data Path. Attacks to the data paths between rich applications and secure tasks fall into three main categories: channel hijacking, man-in-the-middle (MITM), and denial of service (DoS). Since the SMC call has to stem from a privilege mode, the kernel will always lead the rich-secure communication. This represents a threat since an attacker with root access to the system could modify the kernel at run-time to intercept a data path. Examples of possible attacks include: superseding the return of a secure task, manipulating a rich application's internal state and/or memory (e.g., by using *ptrace*), or attempting a return-oriented programming (ROP) attack. These attacks are only viable if root can inject code to the kernel and access the system's main memory. However, if loadable kernel models such as Linux's LKMs, and access to kernel and physical memory (*/etc/kmem* and */dev/mem* respectively) are disabled, these attacks are not possible. We consider that in the context of mobile devices it is not common to dynamically load and unload kernel modules, and therefore enforcing a static kernel at run-time is a fair compromise. Indeed, popular mobile OSs such as Android are beginning to take similar steps towards limiting kernel functionality; from Android 4.3 debugging tools such as *ptrace* are restricted by SELinux [10], making application code more deterministic and resilient to manipulation and hijacking. If we assume that the OS image that is verified at boot time can be trusted, the fact that the kernel cannot be modified at run-time allows us to guarantee that once a system call is being processed in kernel space, an attacker would not be able to tamper with the secure data paths that the kernel establishes with the secure area. This technique has been used before to protect the kernel from a malicious root [25] [14]. Under this scenario, if rich applications make use of secure system primitives to evaluate the running system and access peripherals securely, and secure tasks to access sensitive data and communicate with their IT infrastructure, channel hijacking and MITM attacks can be prevented. Finally, we consider that preventing DoS attacks when using TEEs is a challenge, and an interesting topic for future research.

Hardware attacks. TrustZone is not tamper-resistant, and while SEs are tamper-resistant, they are not tamper proof. The assumption here should be that

[8] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-3051
[9] http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html
[10] http://lwn.net/Articles/491440/

with enough time, money and expertise and attacker could steal the secrets in the SE by means of a sophisticated physical attack (e.g., a laboratory attack). Having physical access, such a sophisticated attack to reveal the secrets in the SE would mean to either break the security methods (e.g., lock screen) of the running and trusted OS or to imitate a trusted system state to the SE. In the second case, the adversary would need to remove the SE in order to be able to bypass the TZPC, and directly send the boot hashes of the original trusted system to the applet. The SE applet does not have the capability to detect the malicious source of the boot hashes and would therefore grant access to the sensitive data.

7 Conclusion

In this paper, we introduce a two-phase boot verification for mobile devices. The goal is to give users the freedom to choose the OS they want to run in their mobile devices, while giving them the certainty that the OS comes from a source they trust. We extend this certainty to running applications, which can verify the environment where they are executing. This is a first step towards usage control. By not locking devices to specific software, users can switch OSs depending on their social context (e.g., work, home, public network). This protects user's privacy, as well as service providers from untrusted devices. We contemplate this as a necessary change in the way we use mobile devices today, and a natural complement to multi-boot virtualization architectures like *Cells* [2]. One device might fit all sizes, but one OS does definitely not. Finally, since our approach is based on off-the-self hardware, it can be implemented in currently deployed mobile devices.

Acknowledgements. This work has been carried out within the scope of u'smile, the Josef Ressel Center for User-Friendly Secure Mobile Environments. We gratefully acknowledge funding and support by the Christian Doppler Gesellschaft, A1 Telekom Austria AG, Drei-Banken-EDV GmbH, LG Nexera Business Solutions AG, and NXP Semiconductors Austria GmbH.

References

1. *The Privacy Engineer's Manifesto*. Number p. 242 - 243. Apress, 2014.
2. J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 173–187. ACM, 2011.
3. W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *Symposium on Security and Privacy*, pages 65–71, May 1997.
4. ARM Security Technology. Building a secure system using trustzone technology. Technical report, ARM, 2009.
5. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

6. C. Doctorow. Lockdown, the coming war on general-purpose computing.
7. M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference*, pages 305–319, 1989.
8. J. González and P. Bonnet. Towards an open framework leveraging a trusted execution environment. In *Cyberspace Safety and Security*. Springer, 2013.
9. J. Goodacre. Technology preview: The armv8 architecture. white paper. Technical report, ARM, 2011.
10. J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.
11. S. Höbarth and R. Mayrhofer. A framework for on-device privilege escalation exploit execution on android. *Proceedings of IWSSI/SPMU (June 2011)*, 2011.
12. M. Hölzl, R. Mayrhofer, and M. Roland. Requirements for an open ecosystem for embedded tamper resistant hardware on mobile devices. In *Proc. MoMM 2013: International Conference on Advances in Mobile Computing Multimedia*, pages 249–252, New York, USA, 2013. ACM.
13. S. Khan, M. Nauman, A. Othman, and S. Musa. How secure is your smartphone: An analysis of smartphone security mechanisms. In *Intl. Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec '12)*, pages 76–81, 2012.
14. S. T. King and P. M. Chen. Backtracking intrusions. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 223–236. ACM, 2003.
15. M. La Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *IEEE Communications Surveys Tutorials*, 15(1):446–471, 2013.
16. H.-c. Lee, C. H. Kim, and J. H. Yi. Experimenting with system and libc call interception attacks on arm-based linux kernel. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 631–632. ACM, 2011.
17. S. Liebergeld and M. Lange. Android security, pitfalls and lessons learned. *Information Sciences and Systems*, 2013.
18. G. Madlmayr, J. Langer, C. Kantner, and J. Scharinger. *NFC Devices: Security and Privacy*, pages 642–647. 2008.
19. Mobile Phone Work Group. TCG mobile trusted module sepecification version 1 rev 7.02. Technical report, Apr. 2010.
20. S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.
21. J. Rouse. Mobile devices - the most hostile environment for security? *Network Security*, 2012(3):11–13, 3 2012.
22. Trusted Computing Group. TPM main specification version 1.2 rev. 116. Technical report, Mar. 2011.
23. Unified EFI. UEFI specification version 2.2. Technical report, Nov. 2010.
24. T. Wu. The secure remote password protocol. In *Proc. of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, Nov. 1998.
25. G. Wurster and P. C. Van Oorschot. A control point for reducing root abuse of file-system privileges. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 224–236. ACM, 2010.