Secure Containers in Android: the Samsung KNOX Case Study

Uri Kanonov
School of Computer Science
Tel Aviv University
urikanonov@gmail.com

Avishai Wool
School of Electrical Engineering
Tel Aviv University
yash@eng.tau.ac.il

ABSTRACT

Bring Your Own Device (BYOD) is a growing trend among enterprises, aiming to improve workers' mobility and productivity via their smartphones. The threats and dangers posed by the smartphones to the enterprise are also ever-growing. Such dangers can be mitigated by running the enterprise software inside a "secure container" on the smartphone. In our work we present a systematic assessment of security critical areas in design and implementation of a secure container for Android using reverse engineering and attacker-inspired methods. We do this through a case-study of Samsung KNOX, a real-world product deployed on millions of devices. Our research shows how KNOX security features work behind the scenes and lets us compare the vendor's public security claims against reality. Along the way we identified several design weaknesses and a few vulnerabilities that were disclosed to Samsung.

CCS Concepts

•**Security and privacy → Mobile platform security;**

1. INTRODUCTION

1.1 Background

The wide range of possibilities provided to us by our smartphone is invaluable to our work place. Being available 24/7, having the ability to rapidly respond to e-mails, to open and edit documents, scheduling meetings, and attending video conferences regardless of our physical location, are all work-related activities. Such a setting in which the work place allows (and even encourages) the user to work from her personal phone is often referred to as Bring Your Own Device (BYOD). The primary reasons for supporting BYOD [25] are keeping employees satisfied (as they can use their own device), mobile and productive (work from anywhere). Surveys of mobile security issues [29] have identified multiple problems enterprises face when dealing with BYOD. These include security policy enforcement, stolen or lost devices containing sensitive data, data confidentiality and integrity when stored on or accessed from the device. In particular the following threats are prevalent:

Untrusted Networks In unencrypted wireless networks, attackers may eavesdrop on the communication or tamper with it via a man-in-the-middle (MITM) attack.

Loss or Theft A prominent risk to mobile devices is loss or theft, giving an attacker physical access to the data stored on the device. Lack of strong encryption places the data in imminent danger.

Malware Android has been the target for malware from the very beginning, reaching 97% of all mobile malware in 2014. E.g., [22] describes how Chinese hackers managed to upload malicious applications to Google Play, with over a million downloads. Once malware is installed, it may be able to collect sensitive data stored on or generated by the device.

Security threat mitigation for mobile devices, and BYOD-engaged devices in particular has been the target of extensive research, resulting in a wide range of solutions. In this paper we focus on a prominent family of solutions for Android, namely "secure containers". These are isolated environments that provide secure storage of data, allow for confined execution of applications and controlled management of resources.

1.2 Related Work

There are multiple BYOD security solutions, each taking a different approach and subsequently providing different levels of security. In this section we review representatives from each category of solutions, focusing on the advantages and shortcomings of each one.

Policy: In [38] the authors recognize that insufficient device management and security policies as well as inter-application data leakage are among the top failures in BYOD security. As a mitigation they propose a security framework including a Mobile Device Management (MDM) system for policy enforcement, device provisioning and incident response in case of detected threats. A concrete solution is MUSES [21]. MUSES is a self-adapting system, utilizing data mining and machine learning with sensors such as connectivity, mail, files, location and root detection. MUSES's key downsides are it's impact on the user's primary environment and vulnerability to root and kernel exploits. Also, MUSES relies on applications directly requesting permission for its policy enforcement.

Root-based: DeepDroid [37] is a custom instrumentation tool, running as the *root* user, allowing the implemen-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPSM'16, October 24 2016, Vienna, Austria

© 2016 ACM. ISBN 978-1-4503-4564-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2994459.2994470

tation of an enterprise fine-grained security policy in addition to Android's built-in permission mechanism and the Mandatory Access Control (MAC) mechanism provided by SEAndroid. DeepDroid's approach has several limitations: (1) The *root* access requirement implies running on a rooted device, thus exposing it to additional dangers. (2) Root or kernel exploits can disable DeepDroid. (3) The management is done on the user's primary work environment.

Secure Containters: A different approach is proposed by Cells [14], utilizing on-device virtualization. The idea is to run multiple "virtual phones" (VP) on a single physical phone, running each VP under its own namespace, isolating its applications and data from other VPs. Cells achieves a significant degree of isolation between VPs—but not as well as can be enforced by SELinux.

Another secure container solution is one designed by Google: "Android for Work" [24]. This solution utilizes the "multiple user" mechanism added in Android 5.0 to create a separate user for a work environment. This allows some isolation of the work area while maintaining the ability to share data with the personal environment.

The Achilles heel of all the solutions we mentioned so far is that a kernel exploit will compromise the security of the entire system.

Hardware-backed: To our aid comes ARM TrustZone [16], helping to move the "root of trust" further away from the attacker. TrustZone is a separate environment that can run security dedicated functionality, parallel to the OS and separated from it by a hardware barrier.

Utilizing the TrustZone, in DroidVault [28] the authors present a security solution for applications that want to store and manipulate sensitive data on the device shielding it even from a compromised kernel. This solution relies on storing the data in encrypted form on the filesystem and manipulating the unencrypted data only in the TrustZone. However, the limitation of DroidVault is that it is relevant only for applications that have a clear cut line between secure and insecure functionality.

Another promising solution, which is the focus of this paper, is the Samsung KNOX [34], a secure container framework built into Samsung's Android-based devices. KNOX, being of vendor origin, has powerful capabilities and protection from the environment (both software and hardware), whose "root of trust" is the ARM TrustZone. However, KNOX is primarily a closed-source system and its architecture is not well documented in the open literature. One of our goals in this this work is to discover and describe the KNOX security architecture. KNOX has been the target of some security research such as [15], exposing security issues relating to password storage. In another work [36], Ben-Gurion researchers have found a vulnerability relating to security of Data in Transit (DIT) but have not published any technical details.

1.3 Contributions

In this work we present a systematic assessment of security critical areas in design and implementation of a secure container for Android through a case study of a real-world system which is deployed on millions of devices: Samsung KNOX. KNOX is a delicate combination of technologies, consisting of multiple components whose integration is Samsung's answer to BYOD security. Our research, backed by extensive reverse-engineering, compares the vendor's security claims to reality, shows how KNOX works behind the scenes and uncovers several design weaknesses. We also discovered some critical vulnerabilities in KNOX and presented practical attacks exploiting them. Our results emphasize the inherent and fundamental pitfalls in the secure container paradigm. Finally, we contrast KNOX 1.0 with the most recent version of KNOX: we show how the latest KNOX improves security— while also making security sacrifices in favor of user satisfaction.

Our findings were disclosed to Samsung in December 2015 [35] and have been identified as CVE-2016-1919 [11], CVE-2016-1920 [12] and CVE-2016-3996 [13]. In accordance with Samsung's request we delayed publicly disclosing the vulnerability described in sections 4.1.2 and 5.2.1 to provide a suitable time-frame for releasing a patch.

The rest of the paper is organized as follows. In section 2 we provide some background material. Section 3 presents our findings about the inner working on Samsung KNOX 1.0. Section 4 details the vulnerabilities we uncovered in KNOX 1.0 and ways to exploit them. Section 5 presents a review of KNOX 2.3 and how it affects our attacks. Finally we present our conclusions in section 6. Details ommitted due to space limitations can be found in our technical report [27].

2. PRELIMINARIES

2.1 ARM TrustZone

Trusted Computing has been the target of much research in the PC world with the purpose of achieving a Trusted Computing Base (TCB), allowing the running of applications in a secure verified environment. ARM TrustZone [16] is the realization of a similar concept in the mobile world through the creation of a Trusted Execution Environment (TEE). It is a set of security oriented extensions implemented by the processor starting with Cortex-A8 and onwards. As stated by ARM [16], TrustZone is meant to withstand software-based attacks (root exploits, kernel exploits) and simple (low-cost) hardware attacks, such that an attacker can attempt at home, e.g., by using a JTAG connection.

The TrustZone specification dictates that each physical processor core is separated into two "worlds", a *normal* world and a *secure* world. The normal world is designated for running the "Rich OS" e.g., Android, whereas the secure world is meant to host a security-centric OS designed for running dedicated applications known as "Trustlets". The same processor can execute both worlds in a time-sliced manner. Each world has its own set of resources (such as flash, memory, CPU caches, etc.). Only the secure world can access both its own and normal world resources (RAM and disk) but not vice versa.

Switching to the secure world can be done by either a dedicated assembly instruction (Secure Monitor Call - SMC) or an interrupt that's configured to be handled in the secure world. The secure world has full MMU support as well as its own user and privileged modes. This allows for concurrent and isolated running of multiple trustlets.

Multiple proprietary secure world OS implementations exist, the most popular of which is QSEE (Qualcomm Secure Execution Environment) [1].These operating systems, although more compact than a traditional OS, are still vulnerable as shown by security researchers [31]. QSEE for instance, has already multiple reported vulnerabilities [30,

18]. A similar counterpart of ARM TrustZone in the PC world is the Intel Software Guard Extensions (SGX) [20].

2.2 Samsung KNOX

KNOX is Samsung's answer to BYOD security threats. Samsung announced KNOX in early 2013, starting with version 1.0.0 [33]. This version has been deployed in Android 4.3 on popular devices such as Galaxy S3 and S4. The most recent version is 2.3, available on more advanced devices such as the Note 3 and onwards.

KNOX belongs to the "secure container" family, providing a secure environment alongside the user's personal environment. KNOX allows running enterprise applications in an isolated environment allowing to control them and configure the environment using MDM APIs. KNOX's "root of trust" is its secure boot sequence, relying afterwards on runtime protections running inside the ARM TrustZone augmented by SELinux in Android.

In this section we provide highlights of the what's known about KNOX from Samsung's whitepapers [33, 34] augmented by information received from Samsung via personal communication [35]. In-depth analysis of KNOX 1.0 and KNOX 2.x resulting from our research is provided in sections 3, 4 and 5.

2.2.1 Architecture

Samsung KNOX has a multi-tier security architecture, in which each layer is secured by its predecessor. Looking top-to-bottom, the layers are:

SEAndroid Android itself and KNOX Applications in particular are protected by a fine-grained security policy enforced by SELinux. The policy protects applications from each other, isolates KNOX applications from user applications and partially mitigates certain attacks. Each application process has a "context" whose limitations (e.g., which files it can access, which processes it can communicate with) are defined by the SELinux policy. This layer's security depends on the integrity of the kernel and the security policy stored on disk.

TIMA On KNOX-enabled devices, the ARM TrustZone runs dedicated security applications whose purpose is ensuring the integrity of the Android kernel at runtime (thus ensuring SELinux's integrity). These security applications are a part of the TrustZone-based Integrity Measurement Architecture (TIMA), further described in section 2.2.3.

TrustZone TIMA relies on the protection and isolation of the TrustZone's secure world from the normal world. The protections are hardware-based (owing to the ARM processor) and software-based (narrow interface to the TrustZone, unlike the Linux kernel).

Secure Boot The initial integrity of the code running within the TrustZone and Android's Linux kernel comes from a process of secure boot, where each step in the boot chain cryptographically verifies the next step. The chain starts from the initial bootloader, fused into the ROM, being the initial root of trust. Additional details are provided in section 2.2.2

KNOX's Trusted Computing Base (TCB) is first and foremost SEAndroid. It assumes the enforcement of the SELinux policy and the lack of a malicious kernel or root user. All of the aforementioned security layers operate together to ensure the validity of this TCB to allow KNOX to execute in a safe environment.

In addition to Android's disk encryption, KNOX applies additional encryption to data stored within the secure container, based on a user-provided password. To further safeguard KNOX data, sharing of data between the personal environment and KNOX container is blocked except for select instances like contacts and calendar events that are dependent on user configuration.

Additionally KNOX provides an extensive VPN framework intended for enterprise applications, as well as a rich set of MDM APIs allowing IT admins an easy and effective method of managing the KNOX container.

Finally, in KNOX 1.0 the only applications that may be installed inside KNOX are those signed by Samsung, and downloaded from a dedicated application store. In contrast, in KNOX 2.x any application can be installed within the limitation of whitelists or blacklists set by each organization's IT admins.

2.2.2 Secure Boot

One of the unique features of KNOX is its Secure Boot sequence. The boot starts from the primary bootloader, burned into the ROM. It loads the secondary bootloader from flash, cryptographically verifying that it is of Samsung origin. The secondary bootloader verifies and loads the secure world OS. The secure world in turns runs TIMA which verifies the integrity of the kernel. If any of the boot components have been tampered with (e.g., by flashing a custom firmware) the device is deemed compromised, causing the "KNOX Warranty Bit" [32] to be turned on. The warranty bit is implemented via a hardware electronic fuse (eFuse) [26], preventing the possibility of a rollback. Moreover, hashes of all loaded components as well as failures to verify components in the boot chain are stored in secure world memory to support device attestation

The purpose of the "KNOX Warranty Bit" is not only for warranty as its name might suggest. If it has been turned on, the device will refuse to create and / or open any KNOX container from that point onwards. The rationale is that KNOX suspects the device has been tampered with and doesn't want to provide a potential attacker access to the user's sensitive information. The criterion for setting the warranty bit is the detection of irreparable alteration to the device. A primary example is flashing the device with a custom firmware (not signed by Samsung). Other scenarios not involving persistent modification in a way that cannot be isolated or recovered from (e.g., like runtime *root* exploits) fall under the responsibility of *TIMA* (see below).

An additional feature of the Secure Boot chain is *dm-verity*, a hash-based verification of critical OS components. Its purpose is to prevent persistent rootkits from getting a foothold in the Android environment. See details in [27].

2.2.3 TIMA

The TIMA is a major security feature of KNOX providing runtime protection. It is a set of trustlets, running within the TrustZone that provide the basis of a secure boot, ensure the system's integrity at runtime and provide security critical services.

Periodic Kernel Measurement (PKM) is a TIMA component that periodically performs validations on the kernel code and data (e.g., that SELinux hasn't been turned off). The precise nature of checks performed by *PKM* is not documented.

Real-time Kernel Protection (RKP) [17] is the core of KNOX's runtime security, guarding against kernel corruption at run-time. The main tasks of RKP include allowing modifications of the page tables only within the secure world, leaving them read-only in the normal world; Ensuring that kernel code pages are never mapped as writable; Never mapping kernel data pages as executable; Preventing double-mapping of kernel memory pages (in particular to user-space); Mapping all user-space memory pages as Privileged eXecute Never (PXN); and transferring control over user process credential structures to the secure world, making them read-only in the normal world. These memory defences together shield the sensitive areas that are the targets most kernel exploits to date.

The way *RKP* enforces its protections on the kernel is by embedding *SMC* calls (see TrustZone section 2.1) in key functions in the kernel code. A similar mechanism to *RKP*, but with less features is presented in Sprobes [23].

Anomalies detected by both *RKP* and *PKM* are logged, followed by an immediate reboot of the device without setting the warranty bit. *RKP* has been available starting from certain models of the Galaxy Note 3. The runtime protections as well as *dm-verity* are enabled if a KNOX license is present but may be enabled even without it depending on the device [35].

3. KNOX 1.0 ARCHITECTURE UNVEILED

Based on Samsung's whitepapers, KNOX looks like a very promising security solution for BYOD. In this section we describe our own observations regarding KNOX 1.0. We present previously unpublished discoveries regarding its design and implementation and review it from a security standpoint. All of our findings were uncovered through comprehensive reverse-engineering of KNOX components and are later utilized in a wide range of attacks on KNOX 1.0 described in section 4. Complementary analysis of KNOX 2.x is presented in section 5.

3.1 Research Environment

We combined both static and dynamic analysis of KNOX 1.0.0 on two Samsung devices running Android 4.3:

SGS3 GT-I9305 Kernel 3.0.31, build JSS15J.I9305XXUEML8

SGS4 GT-I9505 Kernel 3.4.0, build XXUEMJ5.CCOM

Given that KNOX is closed-source, our static analysis had to be performed by pulling relevant binaries from the device and reverse-engineering them. We disassembled native libraries using "IDA Pro" [2]. As for the Java in Android, the byte-code comes in the form of Dalvik Executables (.dex), run by the Android's Java runtime engine: the Dalvik VM. We often encountered .odex (optimized dex) files, which we converted to .dex using "Universal Deodexer" [3]. The .dex files we converted to .jar files using "dex2jar" [4] and finally disassembled to Java code using "jd-gui" [5]. Oddly enough, the resulting Java code wasn't obfuscated.

Performing dynamic analysis (runtime modification of code and placement of hooks) required root privileges in order to bypass Android and KNOX's security features. The Galaxy S3 in our possession was initially rooted by flashing a custom firmware, setting its warranty bit, making KNOX unusable. Indeed when attempting to run KNOX we encountered an error stating "Your device is not authorized to enter Samsung KNOX mode". This drew us to search for an alternative device to experiment on, the Galaxy S4. The S4 was rooted by a personally modified version of *SafeRoot* [6], which utilizes a kernel exploit from CVE-2013-6282 for privilege escalation to root. This method was not detected by KNOX, leaving KNOX fully functional and allowing us to run alongside it as root. Our modification to the basic *SafeRoot* included removing functionality in charge of disabling KNOX. This was necessary to prevent the rooting process from tampering with our research environment.

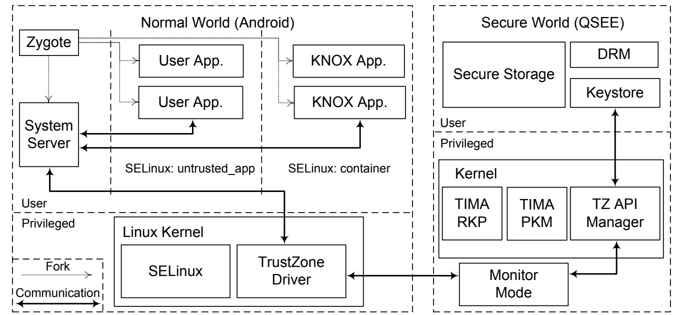

Figure 1: The KNOX Architecture

Once we had root access without tripping the KNOX warranty bit, our dynamic analysis was performed using *Xposed* [7], a framework allowing the modification of Java code at runtime. Moreover, as we advanced in our research, we leveraged *Xposed* to blind KNOX on the Galaxy S3, returning it to full functionality while retaining our root privileges (more on this in section 4.2.4).

3.2 Application Architecture

3.2.1 The KNOX Architecture

The KNOX whitepapers create the impression that KNOX applications run in a separate "container" having nothing in common with regular applications, but we discovered that things are somewhat different. Let us recap how normal applications are run in Android. On boot, the *init* process starts *Zygote*, the initial Dalvik VM. *Zygote* in turn forks and runs the *system_server* which hosts all the OS-provided Java services. Running applications is done by communicating with *Zygote* and asking it to fork, switch to the appropriate SELinux context and run the requested application's entry point. This design makes the *Zygote* process the parent of all application processes and was conceived in order to maximize shared code between applications (by loading common code in *Zygote*). KNOX applications are run precisely in the same manner as normal applications: they are forked from *Zygote* but under a dedicated SELinux context named "container". This context isolates the KNOX applications from other user applications and even hides them when enumerating processes.

In our opinion, relying on SELinux for process isolation is a reasonable design choice. However it does leave a gap: if *Zygote* is compromised, for instance if an attacker manages to run malicious code inside it, the attacker's code would propagate to all KNOX applications and the *system_server*: in fact, this is how *Xposed* works. If SELinux protection is bypassed by some vulnerability, none of KNOX's defence mechanisms are equipped to detect or defend against Java

code injection. We leverage this security gap in several of our attacks (4.2.3, 4.2.4).

Running KNOX is done through a dedicated application named the "KNOX Container Agent", mostly responsible for the login UI. Once the password is entered, the container agent asks the Enterprise Container service (*container_service*) to validate it. The container agent requests the *container_service* to mount the encrypted filesystem, after which it runs the KNOX home screen application. Figure 1 shows the architecture of KNOX 1.0 that we discovered.

3.2.2 Application Wrapping

In KNOX 1.0 the user cannot simply install any application she chooses into KNOX; she must download a previously approved "wrapped" application (signed by Samsung) from a dedicated KNOX app store. In addition to adding a procedure of review and validation for applications, application wrapping is necessary due to a technological limitation. Each application in Android is uniquely identified by its "package"(e.g., com.android.email is the Email application), therefore one cannot install two different applications with the same package. Since KNOX applications run alongside to the normal applications under the same "application namespace", this means that there can't be both "user" and KNOX instances of Email installed on the same device. The solution Samsung selected is wrapping: they repackage the application under a modified package name by prefixing *sec_container_1.* (e.g., sec_container_1.com.android.email). Application wrapping makes it more difficult for developers to develop applications for KNOX 1.0 but on the other hand, it creates a barrier making it more difficult for malware to find its way into KNOX. As a side note, this limitation prevented us from legitimately running our own "applications" under KNOX 1.0.

3.2.3 Shared Services

Another implication of the "side-by-side" design, is sharing the services between KNOX and the user applications through the *system_server*. This central process hosts generic services such as *input_method* (keyboard), *clipboard* and *connectivity* in addition to KNOX-specific services such as *tima* (TIMA service) and *container_service* (Enterprise Container service). Both user and KNOX applications can communicate with *all* of these services and it is the services' responsibility to verify that their client has sufficient privileges before performing any action or relinquishing information. We take advantage of this observation in the two attacks in sections 4.1.2 and 4.1.1.

Lessons Learned: First, *reuse of components*, such as the *system_server* exposes parts of the secure container's critical infrastructure to attackers. Next, given that KNOX utilizes managed (Java) code as a part of its security infrastructure (*system_server*) and allows to run Java-based applications, it must adequately protect the managed-code layer. Otherwise, the secure container becomes highly vulnerable to code-injection attacks, as we consistently show in our attacks (section 4.2). Finally, although *Application wrapping* exists in KNOX 1.0 only due a technical limitation it is an excellent security feature. As already proven by iOS, compared to Android, a thorough process of application review by a trusted party, greatly diminishes the risk of a malware making it to the application store.

3.3 Data Encryption and Password

One of KNOX's primary features is On-device Data Encryption (ODE), whose role is to protect the sensitive corporate data stored within the KNOX container. We discovered that KNOX implements ODE using two filesystem partitions, one for the application data (/data) and one for the sdcard (/sdcard). Each of the two partitions is mounted as an eCryptFS encrypted file system [8]. The secret on which the entire data encryption scheme relies is the user's password.

3.3.1 eCryptFS

The eCryptFS filesystem is file-based, contrary to the stock full disk encryption used by Android (via *dm-crypt*). Additionally, Samsung also uses eCryptFS for sdcard encryption.

The input for eCryptFS is a password 4 to 32 bytes long. In usage scenarios like Ubuntu Home Directory encryption, the encryption password is directly the password provided by the user. However, KNOX uses a more elaborate scheme. In KNOX the encryption password, called the "eCryptFs Key" is a combination of the user's password (minimum 7 chars) and 32 random bytes (denoted by *TIMA key*). We shall discuss how this eCryptFS key is generated in section 4.2.2. To allow the user to change the password without re-encrypting the data, eCryptFS uses a *Data Encryption Key (DEK)* (used to actually encrypt the data) and a *Master Key (MK)* which encrypts the DEK. The MK is derived from the password (eCryptFS Key) and a salt using PBKDF (Password-based Key Derivation Function) [9].

The entire eCryptFS implementation runs inside *vold* (a daemon dedicated to filesystem management), indicating that the DEK is exposed in *plaintext* in *vold*'s memory. In a TrustZone paradigm, one would expect that the DEK would be accessible only to the secure world, protecting it from root attackers in the normal world. In our technical report [27] we elaborate further on KNOX's use of eCryptFS and show ways several attacks a root attacker can attempt to obtain the DEK and gain unlimited access the encrypted data.

3.3.2 TIMA Key

As we mentioned in section 3.3.1, a part of the creation of *eCryptFS key* is the *TIMA key*. The TIMA key, a sequence of 32 random bytes, is generated by the Enterprise Container service during the first KNOX container creation. The user can delete and recreate the container afterwards but the TIMA key will remain the same. Once generated the Enterprise Container service asks the TIMA service to "install" the key. When requested to mount the encrypted file system, the Enterprise Container service "retrieves" the key from the TIMA service, combines it with the password and passes the mount command forward to *vold*. The *install* and *retrieve* operations are forwarded to the TIMA Keystore trustlet running in the ARM TrustZone. The TIMA service will provide the key to any thread running within the *system_server* or process with *system* UID.

In our opinion, a weakness in this procedure is that the TIMA key should have been generated within TrustZone and not in the normal world Android. Not doing so makes the TIMA key available in normal world memory, exposing it to an attacker with root privileges. We leveraged this weakness along with the observations from section 3.2.1 to obtain the TIMA key using *Xposed*.

3.3.3 Inputting the User Password

The ARM TrustZone supports secure I/O which does not pass through the normal world as demonstrated by DroidVault [28]. Although requiring dedicated I/O drivers in the TrustZone, potentially increasing its attack surface, it would provide KNOX with a secure way to obtain the user's password. Without it a transient copy of the user's password resides in the memory of: (i) the KNOX Container Agent, which presents the password textbox; (ii) the onscreen keyboard process that handles the touchscreen input; and (iii) the *system_server* that receives the password via IPC and uses it to create the eCryptFS key.

The only protection around the password that Samsung have put in place is that KNOX only agrees to use the Samsung official keyboard and no other. This is indeed a prudent choice as a malicious keyboard can capture the user input, but other intervention points exist. An attacker with root privileges could intercept the password in each of the aforementioned processes. Case in point, using *Xposed*, we exploited this weakness in order to obtain the user's password from within both the KNOX Container Agent and the *system_server*. This only strengthens our observation regarding necessaity of protection of Java code (see section 3.2.1).

Lessons learned: Use the TrustZone: KNOX 1.0 has a complex data encryption scheme, with many shortcomings and even crucial vulnerabilities. A likely reason for this particular design, is Samsung's desire to achieve maximal component reuse. The main flaws in Samsung's design come from not using the TrustZone enough. Instead we suggest the following design: (i) Read the password from the UI only from within the TrustZone. (ii) Use a well known key-derivation algorithm directly (like PBKDF2) from within the TrustZone to avoid exposing the password and encryption key (DEK) to Android, only providing Android a handle to the key. (iii) Do data encryption and decryption must occur within the TrustZone. Querying and modifying the data should be allowed only through a controlled interface using the TrustZone driver and only given the correct key handle.

3.4 KNOX Warranty Bit and TIMA

The TIMA is a crucial building block in the security structure of Samsung KNOX as is responsible for continued preservation of the system's integrity after boot and maintaining the TCB. The lack of full functionality of TIMA on the Galaxy S4 (*RKP* and *dm-verity* in particular) allowed us to root the device and install the *Xposed* framework, both necessary for our research. This however, only stresses the importance of the TIMA runtime protections. For instance, if *RKP* had been deployed and the process credential structure protection had been enabled (see section 2.2.3), we would not have been able to root the device using *SafeRoot*. This is due to the fact that *SafeRoot* elevates the running process' privileges by modifying the credential structure. Additionally, *dm-verity* would have flagged our replacement of *Zygote*'s binary in the system image, causing the device to enter a "boot-loop".

3.4.1 The KNOX Warranty Bit

Recall that the warranty bit is the indicator whether a device has been compromised or not. It is the TIMA's responsibility to fail KNOX container creation, and any access to an existing container on a device, if the warranty bit has been set. This functionality is present starting from the Galaxy S3 as indicated by the fact that our rooted Galaxy S3 refused to create a KNOX container. By tracking the container creation process, we saw that the API that fails was the TIMA key installation via the TIMA service. This shows that TrustZone TIMA keystore's functionality does depend on the warranty bit. In section 4.2.4 we show how a root attacker can overcome this difficulty quite easily.

Lessons Learned: A sound design decision made by Samsung is the layered security model where each security layer is protected by its predecessor, drawing the initial trust from the hardware. A critical link in the chain is the TIMA which protects the device from various root or kernel exploits and maintains the TCB. We recommend any secure container implementation to base its TCB on hardware support, e.g., like the ARM TrustZone.

4. ATTACKS ON KNOX 1.0

We divide the attacks into two categories, those requiring *root* privileges (i.e., needing an exploitable vulnerability leading to privilege escalation) and those that don't. The vulnerabilities below have been communicated to Samsung in December 2015 [35] and published on *bugtraq* [10] in January 2016 and April 2016 as CVEs [11, 12, 13].

4.1 Root-not-required Attacks

4.1.1 VPN Man-in-the-Middle

In this section we present CVE-2016-1920 [12], a vulnerability which allows a user application running outside KNOX to perform a Man-in-the-Middle (MITM) attack against KNOX SSL/TLS traffic. The vulnerability is the combined result of the following facts: (i) The same certificate store applies to both Android and KNOX applications. (ii) The VPN feature in Android allows an application to register as a VPN provider and route all traffic through it. This involves asking a permission upon installation and VPN connection startup.

The attack scenario requires short-term physical access to an unlocked device (for example the attacker may ask the victim to make a quick phone call from her device).

The attack is performed as follows: (i) Install the malicious application requiring VPN-related permissions. (ii) Install a 3rd party certificate. (iii) Run the malicious application which starts a VPN connection. This will cause a notification to appear with the icon of the malicious application and name of the VPN connection. (iv) Serve forged SSL/TLS certificates while performing MITM.

For as long as the VPN connection is active, all device traffic outside and inside KNOX will be routed through the VPN connection. During this time, in order to intercept SSL/TLS traffic, the malicious application will serve fake website certificates signed with its previously installed 3rd-party certificate. Due to the shared certificate store, any KNOX application relying on a chain-of-trust verification will believe the certificate to be authentic and continue operating as normal, allowing the user to disclose her secret data to the attacker.

The sole requirement for this attack is the ability to manually install an application in the outside Android environment and click away all the warning dialogs. The attack doesn't involve any exploits or require knowing the user's KNOX password. It should be noted that the attack was not tested in conjunction with KNOX's enterprise per-app-vpn feature.

This critical vulnerability demonstrates the danger in sharing resources (in this case, the certificate store) between the user environment and the secure container.

In response to our vulnerability report [35], Samsung informed us that the vulnerability was already known as a limitation of KNOX 1.0 and was corrected in KNOX 2.0.

Another different in nature yet VPN-related attack involving capturing of traffic on Android was demonstrated by BGU researchers in [19]. The novel contributions of our attack are redirection of traffic from KNOX as well as from the user environment and the ability to perform a MITM attack on SSL/TLS traffic.

4.1.2 Clipboard

In this section we present CVE-2016-3996 [13], a vulnerability that allows an attacker to steal the contents of the KNOX clipboard. One of the KNOX proprietary services is *clipboardEx*. It can provide access to either the KNOX clipboard or the Android clipboard according to an inner setting. This setting can be altered externally through a method exposed by the server. This method, like all the other methods, doesn't perform any authentication on the caller, thus allowing anyone to direct *clipboardEx* at the KNOX clipboard at will. Clients communicate with the service through propriatery API containing some client-side security checks. Although these security checks prevent a client from a user application from querying the KNOX clipboard, they can easily bypassed using reflection. Due to the lack of any server-side security checks and the ability to bypass the client-side defences, an attacker without any privileges can switch the *clipboardEx* to the KNOX clipboard and then query it at will. See [27] for full details of the attack.

The attack highlights a crucial "secure container" design pitfall: special attention must be paid to securing any resource used by KNOX and accessible to the user (in this case the service).

An additional interesting fact is that in Android the clipboard data is persistent and is stored under */data/clipboard* (user) and */data/clipboard/knox* (KNOX) accessible only by the *system* user. Both the user and the KNOX clipboard data is stored unencrypted, outside the eCryptFS file system. The fact that the KNOX clipboard is not encrypted unlike the rest of the KNOX data, although insignificant compared to the primary vulnerability, is a security hole in itself. It means that a root or system user can simply read the persistent clipboard data without having to know the user's password.

This clipboard vulnerability was unknown to Samsung when we reported it. Samsung's response was that users should upgrade to KNOX 2.3 [35]. This version, however, is still vulnerable to a modified version of this attack as we show in section 5.2.1.

ADB: In the technical report [27] we also show how an attacker can abuse the *Android Debug Bridge* (ADB) development feature to attack KNOX.

Lessons Learned: The attacks could have been prevented by *limiting the secure container's attack surface.* Concretely, the clipboard attack (4.1.2) by not exposing the *clipboardEx* service to non-KNOX applications and the ADB attack by avoiding communication with processes run as the *shell* user or disabling ADB altogether (the solution chosen by KNOX 2). And again, *sharing resources* between the secure container and the insecure environment is a recipe for disaster. We've seen this through the sharing of the *connectivity* service and the certificate store in the VPN attack 4.1.1. In particular, any resource that can be either monitored (e.g., network data) or modified (e.g., certificate store, GPS coordinate source) must not be shared between the two environments.

4.2 Root-Dependent Attacks

As we've seen, in KNOX 1.0 the scenario where an attacker with *root* privileges co-exists on a device while KNOX is installed and even running is quite possible in the case that TIMA fails to prevent a runtime *root* attack, thus invalidating KNOX's TCB. Such a scenario has been confirmed as possible by Samsung in our correspondence [35], stressing that their aim is to limit the harm whenever possible. In this section we review what a root-privileged attacker can do to exfiltrate or corrupt sensitive KNOX data. It is important to note that, due to SELinux, simply having root privileges is not enough. The attacker must also have a "strong" SELinux context (e.g., that of the init process) that will allow it to bypass the restrictions imposed by SELinux.

4.2.1 Volatile Access to KNOX Data

We've mentioned that logging into KNOX mounts the eCryptFS containers for the application data and sdcard onto */data/data1* and */mnt_1/sdcard_1* respectively. While they are mounted these directories are accessible to the root user and provide read and write access to the KNOX data making the encryption underneath completely transparent.

Moreover, these filesystems remain mounted even when the user logs out of KNOX or is auto-locked after a timeout. We found that once the eCryptFS containers are mounted they remain mounted until device power-off regardless of the KNOX state, exposing the data to a root attacker. Note that the attacker may modify or corrupt the data thereby altering KNOX application functionality.

4.2.2 eCryptFs Key

In this section we present CVE-2016-1919 [11], a vulnerability that allows an attacker to decrypt KNOX encrypted data without knowing the user's password. In section 3.3.1 we described the eCryptFS paradigm in KNOX, but we left out the process of combining the user's password and the TIMA key to produce the *eCryptFS key*.

The key generation algorithm works as follows: Left-pad the password with spaces to 32 chars; Byte-wise XOR the padded password with the TIMA key to produce "random" 32 bytes; Encode the "random" 32 bytes in *base64*; and Return the 32 left-most chars of the base64-encoded string as the *eCryptFS key*;

The problem with this algorithm is that Base64 expands the given input with a ratio of 4:3 (every 3 bytes result in 4 chars). Given this ratio, only the leftmost 24 bytes of the XOR'ed sequence actually affect the eCryptFS key. If the password is up to 8 characters long, the user's password will be completely ignored and only the padded spaces will mix with the TIMA key. Passwords longer than 8 characters can be easily brute-forced on the device.

Moreover, since a root attacker can obtain the TIMA key through the TIMA service, this vulnerability places the *eCryptFS key* and subsequently all of the encrypted data firmly in the attackers hands. Additionally, if the user were to change her password to any other up-to-8-chars sequence,

it wouldn't actually change the *eCryptFS key*, fooling the user into thinking she has somehow changed the way her data is protected.

This attack demonstrates yet another pitfall, for which encrypted data storage, due to its inherent complexity is a prime candidate. We see that integration of several components, each secure on its own (eCryptFs key generation without truncation and the eCryptFs filesystem) can lead to an insecure combination with hazardous results.

In response to our report, Samsung informed us that this vulnerability was identified during an internal security review and was corrected in KNOX 2.0 [35].

4.2.3 Keyboard Sniffing

In this section we describe how a root attacker can sniff the keyboard input to KNOX using Java code injection (see section 3.2.1). As we already mentioned, Samsung has taken a precaution against malicious keyboard applications by limiting KNOX to work only with the Samsung official keyboard. This, however, doesn't defend against root-enabled attackers. In the "input method" architecture in Android the keystroke data passes through several processes: keyboard, requesting application, and *system_server*. KNOX applications use the same service and subsequently the same keyboard process as normal user applications (running outside KNOX). This means that each of these processes, if compromised by an attacker can lead to disclosure or alteration of keystroke data. This scenario obviously applies to root attacks but also to exploitable vulnerabilities in the Samsung keyboard like CVE-2015-4640. Following this attack vector, we have successfully implemented a fully functional KNOX keyboard sniffer. Using the *Xposed* framework we injected Java code to the Samsung keyboard process and inserted hooks into the code flow processing the keystrokes.

We have also demonstrated the ability to take screenshots during the KNOX login process and inside KNOX without alerting TIMA—see [27].

4.2.4 Hiding the Warranty Bit from KNOX

In this section we describe how a root attacker can "blind" KNOX and prevent it from detecting that the warranty bit has been set. We performed this attack on the S3 which has been rooted by flashing a custom firmware, subsequently setting its warranty bit. We revealed in section 3.3.2 that the TIMA key is generated in user-mode and is "installed" and "retrieved" via the TIMA service. Moreover, in section 3.4 we've uncovered that the TIMA key "installation" API call is the one that prevents us from running KNOX on a device whose warranty bit has been turned on. Combining the two facts leads to a solution allowing to bypass the warranty bit check and to obtain a fully functional KNOX.

Using the *Xposed* framework we injected code to the *system_server*. By overriding the *keystoreInstallKey* to always succeed and *keystoreRetrieveKey* to consistently return the same key, we avoided communicating with the TIMA Keystore and supplied KNOX the TIMA key needed to continue creating the container, fully "enabling" KNOX on our rooted Galaxy S3. The success of this attack indicates that in fact no other KNOX APIs test the warranty bit, in particular the SecureStorage API which is crucial to mounting the encrypted file system.

Note that this attack cannot be used to "re-enable" an already active KNOX container after rooting due to inability to obtain the original TIMA key. However, it can be used by an attacker having access to KNOX-designated devices before they reach the end-user. The attacker can root the device and insert a root-privileged backdoor capable of fooling KNOX and launching a wide variety of attacks.

Lessons Learned: We have shown that if a malicious root attacker can get through KNOX's defences he can extract a plethora of sensitive information as well as corrupt the system. The lesson to be learned is that most of the efforts must be placed on not letting the attacker through to begin-with (which is in fact, KNOX's approach). On the other hand, the secure container should use the TrustZone (assuming it at least is not compromised), to discover such attackers whenever possible.

5. KNOX 2.X AND BEYOND

After having thoroughly examined KNOX 1.0, in this section we review KNOX 2.3, the most recent of KNOX to date. To conduct our research we used a Note 3 Model SM-N9005 running Android 5.0 and KNOX Version 2.3 (Kernel version 3.4.0, build number LRX21V.N90055XXUGBOJ6).

5.1 Changes from KNOX 1.0

Samsung released the latest version of KNOX [34] after the release of Android 4.4 relying heavily on its features. KNOX 2.3's front-end application comes in multiple flavours. Besides the default pre-deployed version there is "My Knox" available for download from Google Play and there are others, each version supporting additional features mainly geared towards the enterprise clients.

The concept of "multiple users" was introduced on shareable devices (e.g., tablets) in Android 4.2 with full support for all devices only since Android 5.0. Samsung, requiring the feature for KNOX 2.0 prior to Android 5.0, enabled it only on their smartphones starting from Android 4.4. This feature allows multiple people to use the same physical device, providing each one a separate environment with their own applications. Applications of multiple users can run in the background, while only one user environment is in the foreground. Users are mostly separated from each other but controlled data sharing is supported. Google utilizes this feature for its own "Android for Work" [24] security solution by making the work environment simply run as another user relying on the "user separation" for isolating the work container from the user environment. KNOX 2.3 does the very same thing, running the applications of the KNOX container as a separate user.

KNOX 2.3 suffers from the same design flaw as KNOX 1.0, running all user and KNOX applications in the same "Android" environment, side by side, still all forked from *Zygote* and still sharing the same *system_server*. As in KNOX 1.0, the isolation between (multiple) users and the KNOX container relies on SELinux. In KNOX 1.0 user applications and KNOX applications run under different SELinux contexts, relying on SELinux policy to thoroughly define the boundaries between the two. Instead in KNOX 2.3 all applications run under the same context (*untrusted_app*) but under different SELinux categories which take care of the isolation. This change simplifies the creation of the SELinux policy and leaves less room for errors.

As noted in section 3.2.2, KNOX 1.0 has a mandatory app-wrapping requirement for installing applications inside KNOX due to package name collisions. This restriction was

resolved in Android 4.4 by the "users" feature, clearing the way for KNOX 2.x to allow installation of any application inside the container without the need for wrapping. With the technical limitation lifted, Samsung also made the business decision to no longer force KNOX applications to originate in the Samsung Store; KNOX 2.x allows installing applications from Google Play and other sources. Although this is a leap forward in both productivity and usability of KNOX, it is a setback in terms of security: in KNOX 1.0 Samsung was responsible for what the user could install in her container, serving as an effective malware filter. Now the responsibility has been passed over to the end-user and the IT administrators in the enterprises. In section 5.2.2 we show some of the practical repercussions of this decision.

On the other hand, KNOX 2.3 solves many of the security issues we saw in KNOX 1.0. The "eCryptFS Key" vulnerability was corrected as the password management and encryption were thoroughly revised making much more extensive usage of the TrustZone. The "VPN MITM" attack is also no longer applicable as KNOX 2.3 supports a separate certificate store for the KNOX container as well as separate VPN routing. Moreover, the ADB debugging features were disabled altogether while KNOX is installed on the device. An additional security enhancement comes in the form of separate "keyboard" processes for each user, making it more difficult for attacks on the user keyboard to affect KNOX.

5.2 Attacks

5.2.1 Clipboard

KNOX 2.3 adds a new feature in the form of controlled clipboard sharing. The user may selectively choose to share certain clips between the Android environment and KNOX. This feature is controlled via a policy that can be set by the IT administrators. However, the service architecture still relies on a shared *system_server* process, providing access to the services running within to all users. Apparently in the course of redesigning *clipboardEx* to support clipboard sharing, some security measures were added. When asked to provide clipboard data the service checks against clipboard sharing policy, whether KNOX is active at the moment and which user made the request.

These changes effectively mitigate the "simple" clipboard attack we showed in section 4.1.2. When the attacker requests the clipboard data, *clipboardEx* either returns the user clipboard or refuses to relinquish the KNOX data.

Unfortunately these security improvements aren't enough. Other APIs, critical to our attack remain unprotected. We were able to reproduce our attack by creating a new user activity while KNOX is running in the background. Details of the modified attack are presented in [27].

This attack was successfully tested on the KNOX 2.3 application bundled with the Note 3 as well as on "My Knox" which can be downloaded from Google Play. Merely updating "My Knox" on Google Play is not enough to patch this vulnerability, since it is in the *system_server*, which is part of the core system image, and requires a full system update.

Following our report, Samsung reproduced and isolated the race condition responsible for the vulnerability and starting distributing a patch in March 2016 [35].

5.2.2 Data Exfiltration

Much of Android's protection of application data relies on the permission model. Each application requests a set of permissions to perform the actions it requires. The user can accept or reject these permission requests upon application installation: the typical usual user action is to accept them and continue with the installation. In KNOX 2.3 the user may install an application from Google Play or transfer an application already installed in the Android environment to KNOX. This feature exposes the sensitive data within the KNOX container to dangers from malware by abusing the Android permission mechanism and relying on the blind user acceptance. Note that installing applications inside KNOX does not require root privileges - any user can do it. The only obstacle standing in an attacker's way is the vigilance of the IT administrators, which must effectively use the MDM application white- and black-listing capabilities. In the case of a non-IT managed KNOX container, the responsibility to watch for malicious applications lies directly on the user.

To demonstrate these dangers we've created a "backdoor" application, that when installed inside KNOX is capable of: communicating with an Internet C&C server, downloading extension modules, uploading data exfiltrated from within KNOX, and sending SMS messages.

This demonstrates that applications within the KNOX container are not strongly segregated from each other and IT administrators need to be constantly on the lookout for dangerous applications and use the MDM APIs to monitor and protect their company's devices.

Lessons Learned Special care must be taken to protect the data within the container, not only from outside but also from within. Verification of applications prior to installation by a trusted party can help reduce the risk but not eliminate it entirely. Moreover, and especially in the case that such verification doesn't take place, to better handle malware finding its way into the secure container, we recommend deploying a malware and data leakage detector within the secure container.

6. CONCLUSION

We have presented an extensive security assessment of critical security features in the paradigm of secure containers for Android. Each aspect was demonstrated through a real-world example of a security solution deployed on millions of devices worldwide. Our research has revealed the inner workings on KNOX, contrasting the vendor's security claims with reality. We identified several design weaknesses and some actual vulnerabilities. Through our analysis we presented concrete lessons and guidelines for designing and implementing a secure container. We highlighted the danger of sharing KNOX services with user applications, even in the presence of dedicated security measures. The sharing of services is a two-edged sword: on the one hand allowing simpler design and implementation, but on the other hand creating a constant security threat. We demonstrated the dangers that root and kernel exploits present and the importance of properly mitigating them through a hardware root of trust supported by the ARM TrustZone. We also showed that the TrustZone's mere existence is not enough, requiring proper usage of its features in all surrounding areas to gain the promised security boost. We pointed out the dangers posed by simple applications to information within the secure container as well as the importance of closely tracking application updates. Our findings were shared and discussed in details with the vendor, allowing sufficient time for patches to be distributed. We hope that our work will

help future designers avoid potential pitfalls by highlighting crucial areas, improving future BYOD security solutions.

7. REFERENCES

[1] https://www.qualcomm.com/products/snapdragon/security.

[2] https://www.hex-rays.com/products/ida.

[3] http://forum.xda-developers.com/showthread.php?t=2213235.

[4] https://github.com/pxb1988/dex2jar.

[5] http://jd.benow.ca.

[6] http://forum.xda-developers.com/showthread.php?t=2565758.

[7] http://xposed.info.

[8] http://ecryptfs.org.

[9] https://en.wikipedia.org/wiki/PBKDF2.

[10] http://www.securityfocus.com/archive/1.

[11] CVE-2016-1919. http://www.securityfocus.com/archive/1/537319/30/0/threaded.

[12] CVE-2016-1920. http://www.securityfocus.com/archive/1/537318/30/0/threaded.

[13] CVE-2016-3996. http://www.securityfocus.com/archive/1/538113/30/0/threaded.

[14] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 173–187, Cascais, Portugal, 2011.

[15] Apple Insider. After gaining U.S. government approval, Samsung Knox security for Android found to be "completely compromised". http://appleinsider.com/articles/14/10/23/after-gaining-us-government-approval-samsung-knox-security-for-android-found-to-be-completely-compromised.

[16] ARM. *Building a secure System using TrustZone Technology*. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[17] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *Proc. ACM Conference on Computer and Communications Security*, CCS'14, pages 90–102, 2014.

[18] G. Beniamini. TrustZone exploit in QSEE, part 3. http://bits-please.blogspot.co.il/2015/08/full-trustzone-exploit-for-msm8974.html, 2015.

[19] BGU Cyber Security Labs. VPN related vulnerability discovered on an Android device. http://cyber.bgu.ac.il/blog/vpn-related-vulnerability-discovered-android-device-disclosure-report, 2014.

[20] V. Costan and S. Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/.

[21] P. de las Cuevas, A. Mora, J. Merelo, P. Castillo, P. Garcia-Sanchez, and A. Fernandez-Ares. Corporate security solutions for BYOD: A novel user-centric and self-adaptive system. *Computer Communications*, 68:8–95, 2015. Security and Privacy in Unified Communications: Challenges and Solutions.

[22] Forbes. *Chinese Cybercriminals Breached Google Play To Infect 'Up To 1 Million' Androids*, 2015. http://www.forbes.com/sites/thomasbrewster/2015/09/21/chinese-hackers-beat-google-bouncer.

[23] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the TrustZone architecture. *CoRR*, abs/1410.7747, 2014.

[24] Google. *Android for Work Security white paper*, 2015. https://static.googleusercontent.com/media/www.google.co.il/iw/IL/work/android/files/android-for-work-security-white-paper.pdf.

[25] Holger Schulze. BYOD & Mobile Security report. http://www.slideshare.net/informationsecurity/byod-mobile-security-report, 2014.

[26] IBM. IBM introduces chip morphing technology. http://www-304.ibm.com/jct03001c/press/us/en/pressrelease/7246.wss, 2004.

[27] U. Kanonov and A. Wool. Secure containers in Android: the Samsung KNOX case study. Technical Report arXiv:1605.08567 [cs.CR], arXiv.org, 2016. Available from http://arxiv.org/abs/1605.08567.

[28] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena. DroidVault: A trusted data vault for Android devices. In *19th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 29–38, Tianjin, China, 2014.

[29] J. Oltsik. ESG: a multitude of mobile security issues. *Network World*, 2012. http://www.networkworld.com/article/2222813/cisco-subnet/a-multitude-of-mobile-security-issues.html.

[30] D. Rosenberg. QSEE TrustZone kernel integer overflow vulnerability. In *Black Hat USA conference*, 2014.

[31] T. Roth. Next generation mobile rootkits. In *Hack in Paris*, 2013. http://leveldown.de/hip_2013.pdf.

[32] Samsung. KNOX warranty bit. https://www.samsungknox.com/en/blog/about-rooting-samsung-knox-enabled-devices-and-knox-warranty-void-bit, 2013.

[33] Samsung. Whitepaper : An overview of Samsung KNOX (1.0). http://www.samsung.com/es/business-images/resource/white-paper/2014/02/Samsung_KNOX_whitepaper-0.pdf, April, 2013.

[34] Samsung. Whitepaper: An overview of the Samsung KNOX platform (2.x). https://www.samsungknox.com/en/system/files/whitepaper/files/AnOverviewoftheSamsungKNOXPlatform_V1.12_0.pdf, September, 2015.

[35] Samsung Mobile Security Team. Personal communication, 2015.

[36] Wall Street Journal. Samsung phone studied for possible security gap. http://www.wsj.com/articles/SB10001424052702304244904579276191788427198.

[37] X. Wang, K. Sun, Y. Wang, and J. Jing. DeepDroid: Dynamically enforcing enterprise policy on Android devices. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[38] N. Zahadat, P. Blessner, T. Blackburn, and B. A. Olson. BYOD security engineering : A framework and its analysis. *Comput. Secur.*, 55(C):81–99, Nov. 2015.