# TrubiZone

## Securing Critical Mobile Applications for Android Using ARM TrustZone

Tiago Luís de Oliveira Brito
Supervisor: Prof. Nuno Santos

Instituto Superior Técnico,
Av. Rovisco Pais, 1049-001 Lisboa - PORTUGAL
**tiago.de.oliveira.brito@tecnico.ulisboa.pt**

**Abstract.** With the ever-growing number of connected devices worldwide, and with a more conscious and sharing society, mobile devices are becoming interesting data banks. With such an impact in our lives, mobile developers must focus on developing secure and privacy aware applications to protect users and services. Today's mobile operating systems do not offer fully secure methods to support critical applications, such as e-Health, e-voting and e-money, by not taking advantage of secure hardware technology like TrustZone. This paper proposes a new model for the development of critical mobile applications and a system based on TrustZone implementing it.

## 1 Introduction

Mobile devices are becoming the predominant device for simple everyday computing activities. Actions previously performed by powerful desktop computers can now be easily replicated on mobile devices. A recent study [6] from early 2015, with key statistics for the U.S. market, shows that mobile devices, such as smartphones and tablets, dominate digital media time over the Personal Computer (PC), i.e., Internet searches, games and other digital content consumption, with the trend being to continue raising. Due to the prolific use of mobile devices, mobile applications (apps) start to handle more privacy and security sensitive data. Most notably, these apps are handling photos, health and banking information, location and general documentation. However, as mobile devices store and process increasing amounts of security sensitive data, specially from banking and health monitoring applications, they become more attractive targets for data stealing malware.

Parallel to mobile app market's growth, the number of mobile health apps (mHealth) available is increasing rapidly. In 2013, Research2Guidance [25] reported the existence of more than 97.000 mHealth apps across 62 app stores, with the top 10 apps generating up to 4 million free and 300.000 paid downloads per day. According to a report from MarketsAndMarkets [20], this market is expected to grow even further, from a $6.21 billion revenue in 2013 to $23.49

billion by 2018. Because Android is the most attractive mobile platform for malware [18], and because, according to Reuters [1], medical data is more valuable than credit cards, the mobile health sector is an interesting market for attackers.

To prevent negligent development of health care systems, and protect sensitive data from being disclosed, regulatory laws such as the Health Insurance Portability and Accountability Act (HIPAA) have been established. These laws comprise the standard for electronic health care transactions and must be followed by all developers when managing sensitive health data. But regulatory laws are not enough to protect sensitive data, and criminals are motivated by the valuable health care information to perform critical attacks on hospital networks around the world, consequently leaking or stealing health records of millions of patients. In September 2014, a group of hackers attacked UCLA's Hospital network, accessing computers with sensitive records of 4.5 million people. According to Cable News Network (CNN) [2], among the stolen records were the names, medical information, Social Security numbers, Medicare numbers, health plan IDs, birthdays and physical addresses of UCLA's patients.

In the mobile context, data is even more exposed and vulnerable due to the inherent portability of these devices, the sharing of information to third-party advertisers by device manufacturers or mobile app developers, unregulated management of sensitive medical information, specially because regulatory laws such as HIPAA do not account for the mobile sector, and because of security flaws on medical or consumer device software. To protect sensitive data on mobile apps developers rely on mechanisms such as access control mechanisms, application communication monitoring and privacy enhancement systems, either from native Android [3], iOS [4] and Windows [5] or from extensions. These mechanisms rely on ad-hoc Operating System (OS) and application-level methodologies, which in most cases depend upon a very complex Trusted Computing Base (TCB) code, and do not fully enjoy the potential of the hardware of most modern smartphones, by not taking advantage of technology such as ARM's TrustZone.

### 1.1 Objectivos

Restrições

Contribuições

The remainder of this document proceeds as follows. Section 2 highlights the related work on mobile health security, general-purpose mobile security mechanisms and TrustZone-based systems. Section 3 highlights the architecture of

[1] http://www.reuters.com/article/2014/09/24/us-cybersecurity-hospitals-idUSKCN0HJ21I20140924
[2] http://money.cnn.com/2015/07/17/technology/ucla-health-hack/
[3] https://www.android.com/
[4] http://www.apple.com/ios/
[5] http://www.microsoft.com/en-us/windows

TrubiZone. Section 4 highlights the evaluation methodology and implementation followed by Section 5 which describes the work plan for this project. Section 6 concludes this work.

## 2 Related Work

This section presents the related work and characterizes the main contributions of past works and how these contributions helped in the development of this project. This section is organized in three main parts: mobile health security, general-purpose mobile security mechanisms and TrustZone-based systems.

The first part focuses on describing the attack surfaces of mHealth apps, the most common threats and their seriousness, present a few publicly available insecure mHealth apps as well as some compliance recommendations app developers should follow to avoid unnecessary security risks when handling sensitive health information. The second part of this sections describes the state-of-the-art of general-purpose mobile security mechanisms with a particular focus on the Android Operating System. The third part of the related work describes systems developed using TrustZone, a hardware technology available in most modern ARM processors which supports execution of two isolated worlds, a hardware-protected secure world and a normal world, and how these contributions may be helpful in achieving the main goals of this project.

### 2.1 Studies in Mobile Health Security

As discussed above, mobile devices are increasing in number at astonishing rates, and with this growth the mobile market becomes cheap and accessible. This motivates the shift from mainframe systems, located in the facilities of healthcare providers, to apps on mobile devices as well as storage in shared cloud services. This accessibility also motivates the private sector in building more mobile healthcare applications to support both patients and professionals. Thus, the mobile health sector is becoming a competitive market and one which is increasingly handling more sensitive data.

The following subsections describe work done in assessing the current security state of commercial mobile health applications. The first subsection describes a thread taxonomy for mHealth applications by Kotz, David [17]. The second and third subsections describe work done by He, Dongjing, et al. [13] through the analysis of three studies which answer the following questions: what are the potential attack surfaces, how widespread and serious are the security threats to mobile health applications.

#### 2.1.1 Threat Taxonomy for Mobile Health Applications

Kotz, David [17] defines a threat taxonomy for mHealth and categorizes threats in three main categories: *Identity Threats*, *Access Threats* and *Disclosure Threats*.

Identity threats are described as mis-use of patient identities and include scenarios where a patient may lose (or share) their identity credentials, allowing malicious agents to access their respective Personal Health Record (PHR). *Insiders* (authorized PHR users, staff of the PHR organization or staff of other mHealth support systems) may also use patient identities for medical fraud or for medical identity theft, and *outsiders* may be able to observe patient identity or location from communications.

Access threats are described as unauthorized access to PHR and include scenarios where the patient, which controls the data, may allow broader-than-intended access or disclosure of information, *insiders* who may snoop or modify patient data with intents other than improving the healthcare of the patient, and even *outsiders* which, by breaking into patient records, may leak or modify this data.

Disclosure threats include scenarios where an adversary captures network packets in order to obtain sensitive health data. This problem can be mitigated by using strong encryption methods, but even if the network traffic is encrypted it is possible to analyse the traffic to determine its characteristics [31]. The adversary may also use physical-layer, or link-layer fingerprinting, methods to identify the device type and, because the wireless medium is open, an active adversary may inject frames or may selectively interfere (cause collisions) with wireless frames. These methods may enable the adversary to create a man-in-the-middle situation, to use link-layer fingerprinting methods, or to compromise the devices in a way that divulges their secrets.

This work by Kotz, David [17] helped in understanding what threats are inherent to mHealth systems and their development. An explicit set of rules for developing privacy-sensitive health applications was still needed to ease the development process. Avancha, Baxi and Kotz [1] filled in the gap by surveying the literature, developing a privacy framework for mHealth and discussing the technologies that could support privacy-sensitive mHealth systems.

This survey considers essential accounting for privacy in the design and implementation of any mHealth system, given the sensitivity of the data collected. A developer may use the list of privacy properties provided by this article as a check-list to be considered in any design. Furthermore, a set of questions is left open for researchers to improve the efficiency and effect of these properties. It is also mentioned that privacy challenges identified by this article need to be addressed with urgency, because mHealth devices and systems are being deployed now, and retrofitting privacy protections is far more difficult than building them in from the start.

#### 2.1.2 Potential Attack Surface of Commercial Mobile Health Applications

He, Dongjing, et al. [13] analyse several mHealth applications available in Android's app store to contribute to the understanding of security and privacy risk on the Android platform. In the first of three studies described, 160 apps are

**Table 1.** Description of attack surface (taken from He et al. [13])

| **Attack Surface** | **Description** |
|---|---|
| Internet | Sensitive information is sent over the internet with unsecure protocols (e.g. HTTP), misconfigured HTTPS, etc. |
| Third Party | Sensitive information is stored in third party servers |
| Bluetooth | Sensitive information collected by Bluetooth-enabled health devices can be sniffed or injected |
| Logging | Sensitive information is put into system logs where it is not secured |
| SD Card Storage | Sensitive information is stored as unencrypted files on SD card, publicly accessible by any other app |
| Exported Components | Android app components, intended to be private, are set as exported, making them accessible by other apps |
| Side Channel | Sensitive information can be inferred by a malicious app with side channels, e.g. network package size, sequence, timing, etc. |

analysed to find evidence of security threats. From analysing previous literature, seven attack surfaces are determined to be in need of protection. Those seven attack surfaces are shown in table 1. The authors also document that the 160 apps studied target two different audiences: 129 (81.65%) are for patients, 32 (20.25%) are for healthcare professionals and the remaining 3 (1.90%) are targeted for both. Most apps targated for patients (60%) are in the Life Management category followed by apps that manage and synchronize user health information (PHR Management), which occupy nearly half (46.88%) of these apps. These numbers are a good indicator of what data is handled by most commercial mHealth apps available and motivate the choice made to build a PHR Management app as demo app for TrubiZone.

A few examples of vulnerable applications are also revealed during this study. Regarding unencrypted information sent over the Internet, Doctor Online [6] (patients can talk to doctors online) and Recipes by Ingredients [7] send unencrypted sensitive information, including the user's email and password, in clear text. With regard to logging sensitive information, the study pinpoints CVS/pharmacy [8], which logs the prescription refill details from user inputs, including name, email address and store number, and also logs user login credentials in a debug log message. A malicious party can use this information to view user profiles and prescription history, which could support medical identity theft. Finally, this first study concludes with the example of sleep monitoring apps, such as SnoreClock [9] and Sleep Talk Recorder [10], which store user sleep records as unencrypted audio files on external storage. With read permission for the SD card, as well as

[6] https://play.google.com/store/apps/details?id=com.airpersons.airpersonsmobilehealth
[7] https://play.google.com/store/apps/details?id=com.abMobile.recipebyingredient
[8] https://play.google.com/store/apps/details?id=com.cvs.launchers.cvs
[9] https://play.google.com/store/apps/details?id=de.ralphsapps.snorecontrol
[10] https://play.google.com/store/apps/details?id=com.madinsweden.sleeptalk

internet permission, a malicious app can read a user's sleep recordings and even send them to remote servers.

### 2.1.3 Severity of Attacks to Mobile Health Applications

In the second study, 27 of the top 1080 free apps from the Medical and Health & Fitness categories on Google Play were analysed according to their vulnerabilities. From this analysis, three attack surfaces are identified as the most important ones: *Internet*, *Third Party Services* and *Logging*. Only 7 of these 27 apps use the Internet to effectively send medical information over to remote servers. It is important to understand if the information sent over the Internet is protected. To achieve this, the authors captured network traffic and concluded that only 57.1% (4/7) of these apps use encrypted communication and the remaining 42.9% (3/7) send unencrypted sensitive health information. Among the unencrypted contents sent by these 3 apps are emails, usernames and passwords. This study also concludes that 85.7% (6/7) of these apps are hosted, and store the recorded data, on third party servers. This is an economical and scalable solution for mobile applications, but storing sensitive health records on third party servers can have serious implications, mostly due to app users not being aware that their data is being stored on third party servers and because these users are incapable of telling whether this data is encrypted in such a way that hosting companies do not have access to it.

Some health apps use Bluetooth devices to collect personal health information such as heart rate, respiration, pulse oximetry, electrocardiogram (ECG), blood pressure, body weight, body temperature, quality of sleep and exercise activities. Naveed et al. [22] show how a malicious app can stealthily collect user data from an Android device or spoof it and inject fake data into the original device's app, in what is called an external-device misbonding (DMB) attack. One of the 27 apps analysed in this study connects to external health sensors and uses default PIN code 0000, which makes it vulnerable to the DMB attack.

Along with the logging, external storage, exported components and other problems discussed above which represent explicit channels used for attacks, side channels can be exploited by a malicious party to infer sensitive information from apps, even when they are well-designed and implemented. This study mentions an example by Zhou et al. [32] where a correlation between network payload size and the disease condition a user selects on WebMD mobile [11] exists. Network payload size is publicly accessible in Android, which represents a problem when such correlations can be made. Zhou et al. [32] mitigate the problem by modifying the Android kernel to enforce limitations on accessing Android's public resources.

In the third study, another 22 apps, which send information over the Internet, are randomly selected from the same top 1080 apps and audited to understand what information is effectively being sent over the Internet, thus inferring the seriousness of the threats. The conclusion is that, when used as intended, these apps gather, store and transmit a variety of sensitive user data, which includes

[11] https://play.google.com/store/apps/details?id=com.webmd.android

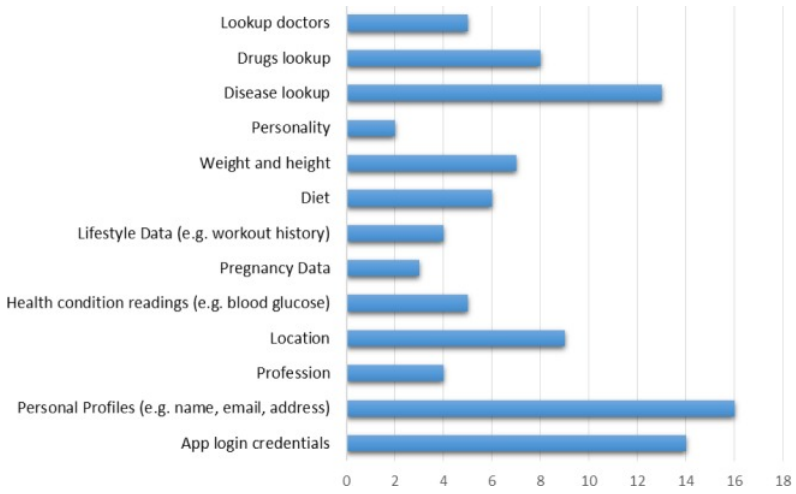


**Fig. 1.** Sensitive information distribution for study 3 (taken from He et al. [13])

at least personal profiles, health sensor data, lifestyle data, medical browsing history and third-party app data (e.g. Facebook account information). Figure 1 shows the distribution of sensitive data in those 22 apps. The consequences of data breaches, information disclosure or tempering with sensitive health data depend on the type, sensitivity and volume of data breached, but it is clear that profiling, medical identity theft and healthcare decision-making errors are all possible. This is why the authors suggest the use of encryption for communication and storage, and encourage developers to create a set of standard security and privacy guidelines that offer a baseline for protection.

The work by He, Dongjing, et al. [13] helped in understanding what are the most common attack surfaces when considering mHealth applications on Android and it also helped assessing the security risks inherent to applications handling privacy-sensitive information. This paper generally describes the state-of-the-art of security in commercial mobile health apps and pinpoints what should be fixed in order to build better and more secure applications for the healthcare industry. It is clear that developers focus much more on building feature-full apps rather than secure applications and this is why it is important to build a framework which allows developers to completely focus on the features they want to provide without having to focus heavily on security.

### 2.2 General-Purpose Mobile Security Mechanisms

This section describes the security mechanisms available for the Android platform, either native or as security extensions, and how these mechanisms can be used to solve some of the problems covered in the previous section. The security mechanisms addressed are divided in four groups: *(a)* access control

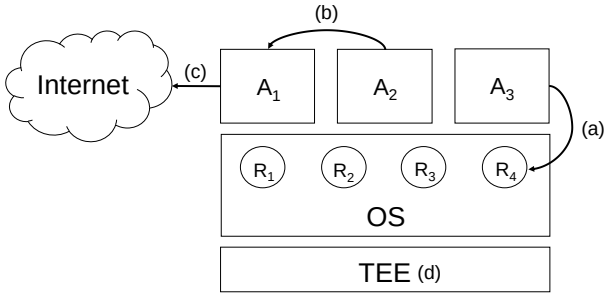


**Fig. 2.** Security Mechanisms Map; A-application, R-system resource

mechanisms, *(b)* application communication monitoring, *(c)* privacy enhancement systems and *(d)* trusted execution environment. Figure 4 maps these four groups to a representation of a possible mobile system.

#### 2.2.1 Access Control Mechanisms

This subsection describes mechanisms and implementations of systems responsible for limiting applications' access to system resources, hence access control mechanisms.

***Access control mechanisms*** are a security model in which subjects (e.g. user, processes, threads, etc.) can perform actions on the system, namely on resources (e.g. files, sensors, etc.), typically called objects. Android follows a type of access control referenced as Discretionary Access Control (DAC), heritage from it's Linux based kernel. In a DAC system the data owner is responsible for this data and thus determines who can access it. In a Linux system, one can imagine a system administrator creating several files and allowing access to these files, according to certain permissions. In this example, a subject with the specified permissions may access the file like its owner intended. In Android we can think of a similar example, since an application can create and store files in the filesystem, thus becoming the sole owner of these files, it can allow access to these files to any other application.

Although Android inherits the DAC from its Linux ancestry, most other resources in Android follow Mandatory Access Control (MAC) policies. In MAC, subjects are much more restricted in determining who has access to their resources. An example of this, from the physical security field, are the levels: confidential, secret and top secret. These labels are the only concept available to define the level of clearance of subjects or to classify data. When a subject attempts to access a classified piece of data, a verification is done to assess if this subject's security level matches (or is above) that of the classified piece of data. Similarly, in Android once a subject attempts to access an object, it triggers a policy evaluation by the kernel, which assesses whether the access may be granted. The advantage of this strict system is its robustness, because

subjects cannot override or modify the security policy. In Android, applications must specify in their manifest the permissions they require at runtime, and after the installation neither applications nor users have any control over the access policies.

Because MAC is robust, several systems were created over the years to extend MAC's access control model to other Android resources. SEAndroid [29] solves problems related to resources complying with the DAC mechanism. The authors ported SELinux [24] to provide MAC at the kernel layer. The kernel was then modified to support a new MAC policy (e.g., filesystem, IPC) and a new middleware layer (MMAC) was created to extend MAC to Android's Binder IPC. TrustDroid [5] extends the MAC mechanism to all the platform's resources (e.g., filesystem which complied with DAC) in order to isolate different domains' sensitive information.

***Permission Refinement*** The Android permission mechanism is a very restrict system. At install time, a list of permissions an application specifies in its manifest file is shown to the user, which is forced into a binary decision, either granting all permissions or quitting the installation. This is an inflexible solution, which makes it impossible for users to have full control of the permissions an application is effectively using at runtime. This inflexibility allows apps to use the device's resources whenever the app wishes without the user's knowledge, possibly with malicious intent.

Over the years, many systems solved some of the problems inherent to permission refinement. APEX [21] modified Android's permission model to allow users to specify the assignment of permissions to each app, both at install and runtime. Permission Tracker [16] allows users to be informed on how the permissions are used at runtime and offers the possibility of revoking these permissions. Furthermore, a user can specify which permissions are of interest so they can be notified of every permission access and decide whether to grant or deny that access.

These systems improve upon the original Android permission model, but require manual configuration by the user. A more useful solution would be to use a context-aware system to handle the permissions at runtime. This way it may be possible to, without manual configuration, restrict permissions to all applications running along side a security sensitive app, thus isolating this application and avoiding possible leaks by shared resources. Several context-aware permission refinement systems exist. Trusted third parties can use CRePE [7] to enforce security policies on another devices whilst, for example, the employees' mobile devices are inside the company, but have no privileges otherwise. Similarly, MOSES [26] enforces domain isolation through the concept of security profiles. MOSES can switch security profiles based on pre-established conditions (e.g., GPS coordinates and time). When switching from a personal (less restrictive) profile to a professional (more restrictive) one, MOSES terminates and prevents the execution of applications not allowed to run in the new enforced profile. Additionally, MOSES leverages TaintDroid [12] to prevent apps from one profile to access data belonging to another.

Both CRePE and MOSES suffer from a device control issue where a third party defines a policy that cannot be revoked by the user. Moreover, a user has no way to deny the enforcement of a third party policy.

***Access Control Hook APIs*** A trend from previous work on mobile security is clearly noticeable: most security extensions require modifying and adding components to the kernel and middleware layers in order to implement new security models. Some frameworks have been built to ease the development process by allowing developers to easily create security models as ordinary apps whilst benefiting from a full callback system capable of notifying the security enforcement applications of accesses and requests to some of the apps' resources of interest.

These frameworks comprise a set of hooks distributed among the kernel and middleware layers, which can be registered by a secure application. When a hook is activated it triggers a callback from the Hook API module, which in turn is forwarded to the app for verification. The app then decides if the operation that triggered the activation of the hook may or may not proceed. The main advantage of frameworks such as Android Security Modules (ASM) [14] and Android Security Framework (ASF) [2] is the flexibility and freedom given to developers in choosing whatever resources to manage.

***Memory Instrumentation*** Memory instrumentation can be divided in two groups: static memory instrumentation and dynamic memory instrumentation. While static memory instrumentation changes already compiled bytecode, dynamic instrumentation patches running processes. Dynamic memory instrumentation is an interesting mechanism as it supports the enforcement of new security models. DeepDroid [30] relies on dynamic memory instrumentation to enforce fine-grained context-aware security policies for enterprise. This is done by patching several system services and tracing system calls.

### 2.2.2 Application Communication Monitoring

Another problematic mechanism in Android is the way applications interact with each other. Some problems were already described in section 2.1 regarding vulnerable mobile health applications, which fall into this category. Two attacks will be described in this section, followed by systems developed to mitigate such attacks on Android. The attacks described are called *Confused Deputy Attacks* and *Collusion Attacks* and are represented in Figure 3. Confused deputy attacks basically consist of unprivileged applications taking advantage of other applications' publicly accessible APIs to perform malicious actions. Collusion attacks consist of an app, which might not have permission to perform an operation, still being able to perform it if there exists another app, belonging to the same developer, installed on the user's device with the permission to perform said operation. This happens because Android's permission system is based on UIDs.

Confused deputy attacks allow applications to use resources without explicitly specifying the necessary permission to do so. A simple example is an app that exposes a service on its API allowing another app to receive photos directly

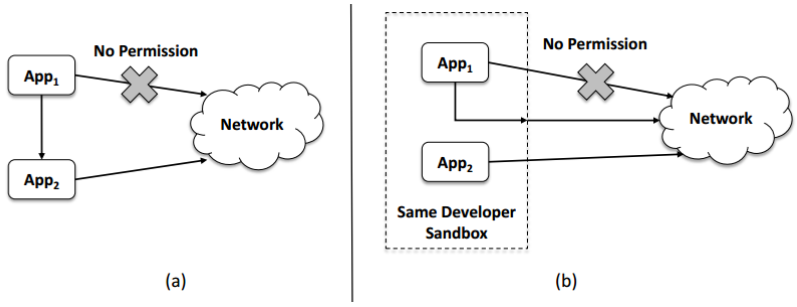


**Fig. 3.** Confused deputy attack (a); Collusion attack (b) - (taken from Duarte [11])

from the device's camera. The second app, which did not specify the camera permission, can access this sensor without the user's knowledge. This problem generally happens because of poor or negligence application development. A system called Saint [23] was created to specify which apps can access the public APIs of another app. Other systems control this communication by extending Android's Binder Inter Process Communication (IPC) mechanism, which is the main form of application inter-communication in Android. QUIRE [9] denies access to an API if in the message exchanged between apps, which contains the full call chain context, the source of the request does not have the necessary permission to access the corresponding data.

Collusion attacks are based on a malicious developer building a legitimate application with, for instance, permissions such as microphone access and persuading the user to install another app with Internet permission. Both apps can collude to leak sensitive audio data through the Internet because Android's permission system is based on UIDs, thus, although the first app does not explicitly request Internet access, this resource is accessible because the developer has Internet access through the second app. XManDroid [4] extends the Android permission model in order to support policies that could constrain the way apps interact with each other. This system prevents data leakage or other types of collusion attacks by developing a graph at runtime with a representation of the apps' interactions, which is then used to regulate app inter-communication.

### 2.2.3 Privacy Enhancement Systems

The most important concern for a privacy-sensitive mobile application is the way data is managed, specially with valuable data such as health records or banking information. Systems like MockDroid [3] and Zhou et al. [33] are extensions to Android's data access control and prevent untrusted applications from accessing sensitive data by allowing users to manually specify application access rights over the system services. These systems even offer data shadowing [12] to

[12] Data shadow is the return of empty or incorrect information instead of the intended data.

the unauthorized apps trying to access it. Other systems, like IdentiDroid [28] focus on protecting the identity of the user. This is done by using an anonymous mobile device state to provide data shadowing and permission revocation techniques that disable the ability of apps to use systems services such as location or International Mobile Equipment Identity (IMEI). This way, if an app is trying to use the devices location, a default location is returned instead of the real one or the application simply cannot access the location service.

Some systems use a different approach to solve the same problem. Dynamic taint analysis systems prevent data leakage by tainting data with a specific mark and then evaluate how this data is propagated through the system. If this data attempts to leave the system the user is alerted. TaintDroid [12] does this by modifying Android's Dalvik VM environment and Binder IPC to track the information within different applications' processes. This system suffers from limitations such as *(i)* tracking a low number of data sources (mainly sensors), *(ii)* performance overheads not tolerable for most mobile environments, *(iii)* the existence of false positives leading to access control circumvention and *(iV)* the incapacity of analysing sensitive information leakage through covert channels.

Despite these limitations, TaintDroid was used by the research community as a building block for other privacy enhancement systems. Kynoid [27] extends TaintDroid to support a bigger number of information sources, and AppFence [15] provides a mechanism to block data made available by the user for on-device use only to be transmitted over the network and allows for high level data to be substituted by shadowed data.

#### 2.2.4 Trusted Execution Environments

***Digital Rights Management*** is a specific access control technology which allows data owners to restrict if and how their data is copied and also how that data is handled once transferred to another device. The Digital Rights Management (DRM) ecosystem is composed of the following entities:

- *User* - human user of the DRM Content
- *Content Issuer* - entity that delivers the content
- *Rights Issuer* - entity responsible for assigning permissions and constraints to DRM content
- *Rights Object* - XML document generated by a Rights Issuer expressing the restrictions associated to the content.
- *DRM Agent* - trusted entity responsible for enforcing permissions and constraints upon the DRM content

To understand how DRM works in the context of mHealth one can suggest a simple example of a PHR mobile health application. In this scenario, the healthcare provider (e.g. a hospital) would be the *content issuer*, and it would use a *rights issuer* to assign the restrictions imposed upon the DRM content, which in this case would be the personal health record of a patient, when this content is transferred to the patient's device. When using DRM, the patient is limited to access the content through a *DRM Agent*.

The Open Mobile Alliance (OMA) developed a DRM standard [10] which defines the format of the content delivered to DRM Agents, as well as the way this content can be transferred from the Content Issuer to the DRM Agent. Android provides an extensible DRM framework, called Android DRM Framework [13], allowing application developers to enable their apps to manage rights-protected content by complying with one of the supported DRM schemes (specific mechanisms, enforced by DRM Agents, to handle particular types of files).

***TrUbi*** is a system developed by Duarte [11] and Costa [8] built on top of ASM [14] allows for flexible system-wide resource restriction, which may be used to isolate privacy-sensitive apps by killing, freezing or revoking permissions to running applications. With TrUbi it is possible to isolate the execution of a critical application from the remaining apps installed on the system, for example, the PHR app could be developed with the following premise. When the PHR app is running, all the other apps are killed and all the resources blocked. The app could then download the health records from the healthcare provider and show the data to the user. When the user exits the application, their data is encrypted with a key generated from a user password, and only then the resources are released for the other applications. The system was completely isolated during the whole process and the data is stored with encryption (and the key, because it is generated from a user password, is not stored on the system). The limitation of TrUbi is based upon the fact that the TCB is comprised of all the Android kernel, Android middleware and extensions such as ASM.

***External hardware security modules*** represent the classic security solution for embedded applications, which is the inclusion of a dedicated trusted element (e.g., a smartcard) that is outside of the main Socket-on-Chip (SoC). On one hand, the main advantage of this solution is that it allows for the encapsulation of sensitive assets inside a physical device specially designed for robust security. On the other hand, the main disadvantage is that smartcards provide only secure processing and storage functions. This means that some operations (e.g., I/O) must rely on software running outside of the security perimeter to provide the desired features. An example where this happens is when a user interacts with the system by entering a Personal Identification Number (PIN). This PIN must be managed by the less secure software outside of the smartcard, making it vulnerable to attacks.

***Internal hardware security modules*** are, contrary to the previous solution, included within the SoC. These integrated modules are usually one of two forms: the first is a hardware block designed specifically for managing cryptographic operations and key storage, and the second is a general purpose processing engine, which is placed alongside the main processor, that uses custom hardware logic to prevent unauthorized access to sensitive resources. This solution has the advantage of being cheaper and offering a performance improvement over external

[13] http://developer.android.com/reference/android/drm/package-summary.html

hardware security modules like smartcards. The disadvantage is that, like the previous solution, the resources protected by the security module will eventually need to be used outside the module it self, thus making these resources vulnerable. Another disadvantage is that this design requires a separate physical processor, typically less powerful than the main processor, to a avoid sharing such an important resource with less critical modules.

***Software virtualization*** is another security mechanism which has been growing in popularity over the past years. In this software security mechanism, a highly trusted management layer called hypervisor runs in privileged mode of a general purpose processor. The hypervisor uses a Memory Management Unit (MMU) to separate several independent software platforms, running each one inside a virtual machine. There are many advantages of using this solution. The first advantage is that a compact hypervisor can be thoroughly tested to ensure, with a high degree of certainty, that software running within one virtual machine cannot influence and attack the execution of others running in parallel. Another advantage is that there is no additional hardware requirement to implement a hypervisor, thus any processor with a MMU can be used to implement this security solution. Lastly, this solution is capable of supporting processing isolation between a secure environment and a full-blown rich operating system running in different virtual machines managed by the hypervisor. If a communication mechanisms is supported by the hypervisor, then a secure pipeline can be established between the two virtual machines.

The main problem with this approach is the isolation provided is restricted to the processor implementing the hypervisor. If all the other resources, mainly system buses, are not managed by the hypervisor, then the protection provided by the virtualization may be bypassed. Managing resources, like Graphics Processing Units (GPUs), can be difficult to achieve without hindering the system's performance.

***TrustZone*** is a hardware architecture that extends the security throughout the system's design. TrustZone's architecture was designed to mitigate the problems discussed above. This is achieved by allowing any part of the system to be made secure, instead of protecting only the assets inside a secure hardware block, thus enabling end-to-end security without exposing resources to a less critical platform.

FALAR DA ARQUITECTURA; HARDWARE E SOFTWARE FEATURES DA TRUSTZONE

### 2.2.5 Summary

After describing and assessing the available technology, one can argue which mechanisms better suit the development of critical mobile applications for Android, such as our PHR app. Some mechanisms are clearly unsuitable for the goals of this project. A summary of the main conclusions for the remaining systems studied is shown in table 2.

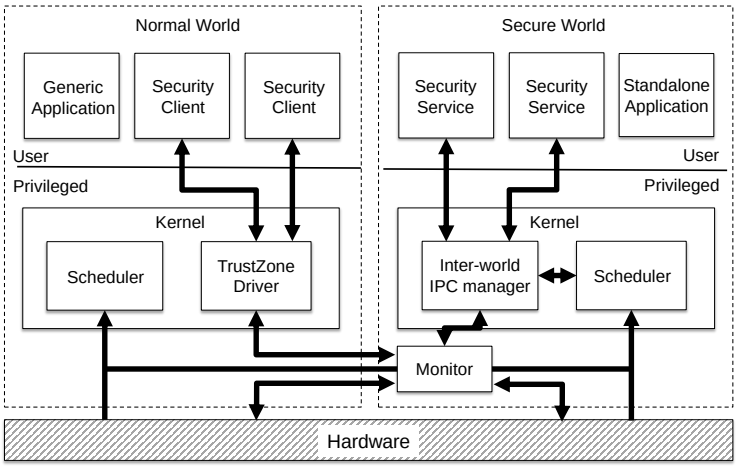


**Fig. 4.** TrustZone's Software Architecture (adapted from [19])

**Table 2.** Comparison between security mechanisms with and without a compromised Operating System (Android).

| Security Mechanism | Trusted OS | | Compromised OS | |
|---|---|---|---|---|
| | Confidentiality | Data Integrity | Confidentiality | Data Integrity |
| Access Control Mechanisms | ✓ | ✓ | ✗ | ✗ |
| App Communication Monitoring | ✗ | ✗ | ✗ | ✗ |
| Privacy Enhancement Systems | ✓ | ✓ | ✗ | ✗ |
| Digital Rights Management | ✓ | ✓ | ✗ | ✗ |
| TrUbi | ✓ | ✓ | ✗ | ✗ |
| External Hardware Security | ✓ | ✓ | ✓ | ✓ |
| Internal Hardware Security | ✓ | ✓ | ✓ | ✓ |
| Software Virtualization | ✓ | ✓ | ✓ | ✓ |
| TrustZone | ✓ | ✓ | ✓ | ✓ |

Although all of the above solutions are suitable for building a secure PHR application, they do not achieve the goals of this project, because all of these mechanisms suffer from the same limitation: both the middleware and the underlying kernel are considered as TCB. This means that if a malicious app is capable of compromising these layers, then the critical PHR application is compromised as well, and the data it protects may be accessible by the malicious party.

The only solution for the problem described is an hardware based security mechanism. As shown in table 2, the embedded security technology, particularly TrustZone, is the only solution which works either with a trusted OS and a compromised OS. For this reason, the next section will focus on describing the state-of-the-art, architecture and functionality of TrustZone technology, as it is the only security mechanism suitable for the needs of this project.

To accomplish the goals of this project, a completely secure flow of execution is needed, this means that privacy-sensitive data must never leave the trusted

security perimeter. A system which does not support secure I/O greatly limits the functionality offered by a PHR application, to the point where it becomes useless. For this reason we exclude both the external and internal hardware security modules as viable solutions for the problem. On a different note, the virtualization solution seems promising, but with all the problems related to sharing resources such as GPUs and other system buses, this security module can be discarded as a solution. For an application to be secure, even when the rich OS is compromised, total isolation is needed, which is not easily achievable using virtualization by it self. TrustZone technology seems to be the most complete system of all the embedded solutions, as it mitigates recurring problems from other mechanisms. Considering this, TrustZone is the most viable solution for our PHR application, because it allows for a complete isolation between the privacy-sensitive data and all the other non-critical logic.

### 2.3 TrustZone-based Mobile Security Systems

FALAR EXCLUSIVAMENTE EM SISTEMAS TRUSTZONE
COMO FUNCIONAM
PONTOS FORTES
PONTOS FRACOS

## 3 Architecture

Your proposed architecture. Can have lots of pictures and bullet points so it is easy to understand.

## 4 Implementation and Evaluation

Explain how you are going to show your results (statistical data, cpu performance etc). Answer the following questions:

- Why is this solution going to be better than others.
- How am I going to defend that it is better.

## 5 Conclusions

Wrap up what you wrote.

## References

1. Sasikanth Avancha, Amit Baxi, and David Kotz. Privacy in mobile technology for personal healthcare. *ACM Computing Surveys (CSUR)*, 45(1):3, 2012.

2. Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android security framework: Enabling generic and extensible access control on android. *arXiv preprint arXiv:1404.1395*, 2014.
3. Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49–54. ACM, 2011.
4. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.
5. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 51–62. ACM, 2011.
6. Comscore. Digital Future in Focus. Technical report, Comscore Inc., 03 2015. http://www.comscore.com/Insights/Presentations-and-Whitepapers/2015/2015-US-Digital-Future-in-Focus.
7. Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Information Security*, pages 331–345. Springer, 2011.
8. Miguel Costa. SecMos - Mobile Operating System Security. Master's thesis, Instituto Superior Técnico, Portugal, 2015.
9. Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, page 24, 2011.
10. OMA DRM. Open mobile alliance digital rights management.(2010). *Retrieved May*, 2, 2011.
11. Nuno Duarte. On The Effectiveness of Trust Leases in Securing Mobile Applications. Master's thesis, Instituto Superior Técnico, Portugal, 2015.
12. William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.
13. Dongjing He, Muhammad Naveed, Carl A Gunter, and Klara Nahrstedt. Security concerns in android mhealth apps. In *AMIA Annual Symposium Proceedings*, volume 2014, page 645. American Medical Informatics Association, 2014.
14. Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium (SEC'14)*, 2014.
15. Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.
16. Michael Kern and Johannes Sametinger. Permission tracking in android. In *The Sixth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies UBICOMM*, pages 148–155, 2012.
17. David Kotz. A threat taxonomy for mhealth privacy. In *COMSNETS*, pages 1–6, 2011.
18. F-Secure Labs. Mobile Threat Report. Technical report, F-Secure Labs., 03 2014. https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf.

19. ARM Limited. ARM Security Technology - Building a Secure System using TrustZone Technology. Technical report, ARM Limited, 04 2009. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
20. MarketsAndMarkets. Mobile health apps & solutions market by connected devices (cardiac monitoring, diabetes management devices), health apps (exercise, weight loss, women's health, sleep and meditation), medical apps (medical reference) – global trends & forecast to 2018. Technical report, MarketsAndMarkets, 09 2013. http://marketsandmarkets.com/.
21. Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.
22. Muhammad Naveed, Xiaoyong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. Inside job: Understanding and mitigating the threat of external device mis-bonding on android. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, pages 23–26, 2014.
23. Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.
24. NSA Peter Loscocco. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track:... USENIX Annual Technical Conference*, page 29. The Association, 2001.
25. Research2Guidance. Mobile Health Market Report 2013-2017. Technical report, Research2Guidance, 03 2013. http://research2guidance.com/.
26. Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. Moses: supporting operation modes on smartphones. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 3–12. ACM, 2012.
27. Daniel Schreckling, Johannes Köstler, and Matthias Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Information Security Technical Report*, 17(3):71–80, 2013.
28. Bilal Shebaro, Oluwatosin Ogunwuyi, Daniele Midi, and Elisa Bertino. Identidroid: Android can finally wear its anonymous suit. 2014.
29. Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.
30. Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS'15). The Internet Society*, 2015.
31. Charles V Wright, Fabian Monrose, and Gerald M Masson. On inferring application protocol behaviors in encrypted network traffic. *The Journal of Machine Learning Research*, 7:2745–2769, 2006.
32. Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028. ACM, 2013.
33. Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, pages 93–107. Springer, 2011.