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 [7] 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 [32] 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 [26], 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 [20], and because, according to Reuters [1], medical data is more valuable than credit card information, the mobile health sector is becoming 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, iOS and Windows 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 Goals

The goal of this project is to fill in the security gap in the mobile application market by proposing *TrubiZone*, a development system with a small, dedicated TCB which, by using ARM TrustZone technology, allows app developers to securely display sensitive content in several formats. With *TrubiZone* the user is guaranteed to access non-compromised data which is isolated from a rich OS environment such as Android. To support the features referenced above, the final implementation must guarantee the following requirements:

Assure Security Policies
TrubiZone must guarantee the fundamental security properties of confidentiality,

[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/

integrity and authenticity, on which every high level security application can be built on.

Trusted User Interfaces
To allow the development of realistic mobile applications, secure user interfaces must be supported, while trying to maintain a small, compact TCB.

Support for General Applications
The implementation should run general applications instead of dedicated and specifically crafted apps. To support this restriction *TrubiZone* must execute applications built using standard trusted execution environment APIs.

Developer Friendly
Developers must be able to easily specify the security properties required and the security-sensitive app logic to transparently run in the trusted environment. This allows developers to focus on the logic and design of said application rather than mobile security mechanisms.

In summary this work expects to contribute with:

- The design of a novel security system, based on TrustZone, for development of secure mobile applications.
- Implementation, on a development board, of a prototype of this new security framework.
- Implementation of a mobile health application using the prototyped framework.
- Assessment of the prototype and mobile health app.

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

2 Related Work

This section describes the related work and characterizes the main contributions from the research community 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. It also shows 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 section 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 support the execution of two isolated worlds, a hardware-protected secure world and a normal world, and how these contributions may be helpful in achieving the 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 threat taxonomy for mHealth applications by Kotz, David [19]. The second and third subsections describe work done by He, Dongjing, et al. [15] through the analysis of three studies which answer the following questions: what are the potential attack surfaces, how widespread and how serious are the security threats to mobile health applications.

2.1.1 Threat Taxonomy for Mobile Health Applications

Kotz, David [19] 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 a 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 malicious intent, 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 [41]. 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.

This work by Kotz, David [19] 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 should use the list of privacy properties provided by this article as a checklist to be considered in any design. 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. [15] 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 were analysed to find evidence of security threats. From surveying previous literature, seven attack surfaces, shown in Table 1, are determined to be in need of protection. The authors also document that most apps targeted 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 [3] (patients can talk to doctors online) and Recipes by Ingredients [4] send unencrypted sensitive information, including the user's email and password, in clear text. Regarding logging, the study pinpoints CVS/pharmacy [5], which logs the prescription refill details from user inputs, including name, email address and store

[3] https://play.google.com/store/apps/details?id=com.airpersons.airpersonsmobilehealth
[4] https://play.google.com/store/apps/details?id=com.abMobile.recipebyingredient
[5] https://play.google.com/store/apps/details?id=com.cvs.launchers.cvs

Table 1. Description of attack surface (taken from He et al. [15])

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.

number, and also logs user login credentials in a debug log message. A malicious party can use this information to access prescription history, which could support medical identity theft. Finally, this first study concludes with the example of sleep monitoring apps, such as SnoreClock [6] and Sleep Talk Recorder [7], which store user sleep records as unencrypted audio files on external storage. With read permission for the SD card, as well as 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.

[6] https://play.google.com/store/apps/details?id=de.ralphsapps.snorecontrol
[7] https://play.google.com/store/apps/details?id=com.madinsweden.sleeptalk

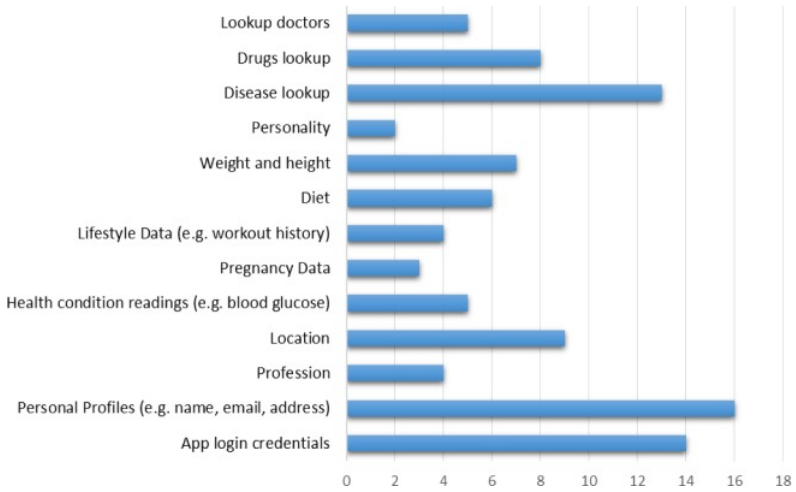

Fig. 1. Sensitive information distribution for study 3 (taken from He et al. [15])

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. [29] 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. [42] where a correlation between network payload size and the disease condition a user selects on WebMD mobile [8] exists. Network payload size is publicly accessible in Android, which represents a problem when such correlations can be made. Zhou et al. [42] 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 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

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

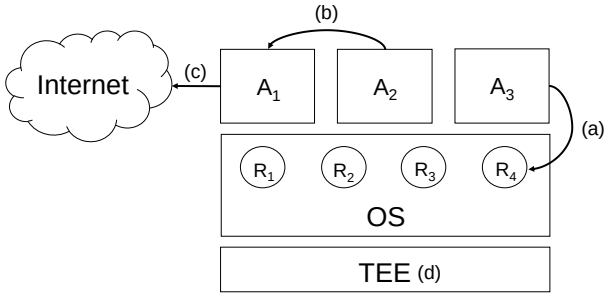

Fig. 2. Security Mechanisms Map; A-application, R-system resource

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. [15] 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 mechanisms, *(b)* application communication monitoring, *(c)* privacy enhancement systems and *(d)* trusted execution environment. Figure 2 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 and sensitive data, 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 Android, 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 are physical security 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 [36] solves problems related to resources complying with the DAC mechanism. The authors ported SELinux [31] 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 [6] extends the MAC mechanism to all the platform's resources 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 [28] modified Android's permission model to allow users to specify the assignment of permissions to each app, both at install and runtime. Permission Tracker [17] 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 [8] 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 [33] 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). Additionally, MOSES leverages TaintDroid [13] 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) [16] 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 [40] 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.

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 it 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 [11] 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 [9], 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).

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 system called Saint [30] 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 [10] denies access to an API if in the

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

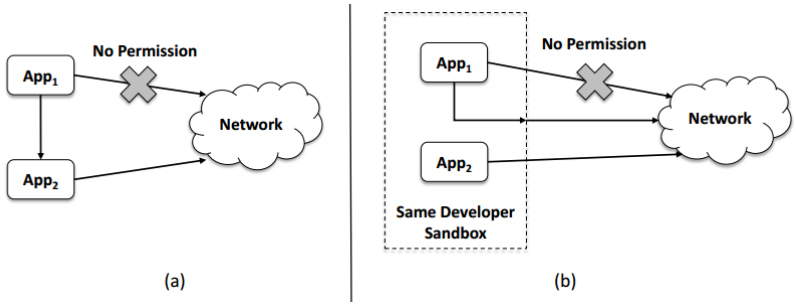

Fig. 3. Confused deputy attack (a); Collusion attack (b) - (taken from Duarte [12])

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 and persuading the user to install a second app with different permissions. Both apps can collude to leak sensitive information by cooperating with each other. XManDroid [5] 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. [43] 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 [10] to the unauthorized apps trying to access it. Other systems, like IdentiDroid [35] 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, such as TaintDroid [13], prevent data leakage by tainting data with a specific mark and then evaluate how this data is propagated

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

through the system. If this data attempts to leave the system the user is alerted. 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.

2.2.4 Trusted Execution Environments

Trusted execution environments are isolated environments that provide a higher level of security than full-blown rich mobile operating systems by guaranteeing code and data to be loaded inside a small protected execution space.

TrUbi is a system developed by Duarte [12] and Costa [9] built on top of ASM [16] 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).

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 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 supports 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 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. This architecture was designed to mitigate the problems discussed above 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.

TrustZone's hardware and software architecture is described in a whitepaper [24] by ARM. In a TrustZone-enabled system, a physical processor provides two virtual cores, one considered non-secure and the other secure, as well as a robust context switching mechanism known as monitor mode. The NS bit sent on the main system bus identifies which of the virtual core performed an instruction or data access. Several software architectures can be implemented on the software stack of a TrustZone-enabled processor, but the most powerful one is a dedicated operating system in the secure world, as shown in Figure 4. This design allows for concurrent execution of multiple secure world applications and services that are completely independent of the normal world environment, thus even if

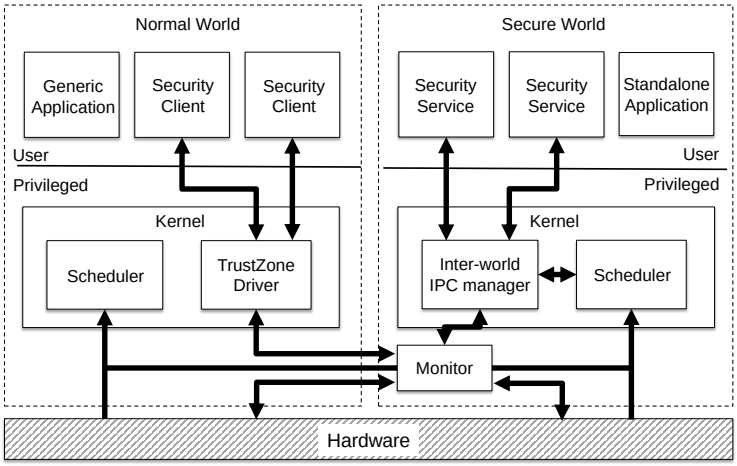

Fig. 4. TrustZone's Software Architecture (adapted from [24])

the normal world is sabotaged, the secure world executes as expected, without being compromised. Moreover, if the secure world kernel is small and correctly implemented, security applications can execute at the same time without needing to trust each other. The kernel design can enforce the logical isolation of secure tasks from each other, preventing one secure task from tampering with the memory space of another.

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. Access control mechanisms and application communication monitoring systems are clearly unsuitable for the goals of this project. These systems may be helpful in developing new access control policies or mitigating vulnerabilities caused by negligent development, but are not sufficient to build privacy-sensitive applications. Privacy enhancement systems focus on detecting and preventing data leakage instead of providing a framework for developers to build privacy-sensitive applications, for this reason these systems do not fulfil the needs of this project. Besides the problems discussed above, all these mechanisms suffer from a critical 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 a critical application is compromised as well, and the data it protects may be accessible by a malicious party. The only solution for the problem described is using a Trusted Execution Environment (TEE). As shown in table 2, the TEE solutions are the only solutions which work either with a trusted OS and a compromised OS.

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	✓	✓	✓	✓

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, both the external and internal hardware security modules were discarded. 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, and because total isolation is needed, which is not easily achievable using virtualization by it self, this security module is not appropriate for this project. TrustZone technology seems to be the most complete system of all the TEE solutions, as it mitigates recurring problems from other mechanisms. For this reason, the next section will focus on describing some state-of-the-art TrustZone-based systems, as it is the only security mechanism suitable for the needs of this project.

2.3 TrustZone-based Mobile Security Systems

The research community has been experimenting with trusted execution environments in order to achieve better solutions for common security problems, such as two-factor authentication or cryptographic key generation. Among the available TEE supported architectures, ARM's TrustZone is the one which has been used the most by researchers and companies world-wide, this is because the deployment potential for these new security applications is tremendous, as most modern ARM processors support this technology.

Several TEE software stack implementations have been designed since the concept was formalized around 2007 by the OMTP standardization forum, which issued a set of security requirements on functionality a TEE should support. The GlobalPlatform [11] organization went a step further by defining standard APIs: on the one hand, the TEE internal APIs that a Trusted Application can rely on, and on the other hand, the communication interfaces that rich OS software can use to interact with its Trusted Applications. It is worth noting that, because the

[11] https://www.globalplatform.org/

TEE threat model assumes that nothing coming from the rich OS is trustworthy, the designer of a TA (Trusted Application) must assume that the rich-OS-side client of the TA may not be legitimate.

When surveying the literature, it is clear that TrustZone-based systems can be divided in two separate groups: General-purpose Frameworks and Runtimes (GPFR), and special-purpose Trusted Applications (TA). GPFR represent tool kits for building secure special-purpose operating systems or runtimes, which allow the development of Trusted Applications. Special-purpose TA are secure applications, built on top of GPFR systems, which are executed in the trusted execution environment to achieve a specific security property. With a more thorough analysis, one can divide each of these categories even further, with regards to trusted user interface (UI). Trusted UI is a TEE feature capable of ensuring that the correct information is displayed to the user and that this information is secure from the risk of, for instance, password logging, as well as allowing logs and statement information to be securely displayed to the user.

For this reason, this section categorizes TrustZone-enabled systems into four subsections: *(i)* GPFR with Untrusted UI, *(ii)* TA with Untrusted UI, *(iii)* GPFR with Trusted UI and *(iv)* TA with Trusted UI.

2.3.1 GPFR with Untrusted UI

Most of the TEE software stack implementations have similar architectures (similar to the one shown in section 2.2.4). This architecture is generally composed of a small trusted kernel running in the isolated secure world of TrustZone-enabled processors, a normal world user space client API and a kernel TEE device driver used to communicate between worlds. OP-TEE [12], TLK[13], TLR [34] and AndixOS [14] are TEE implementations which share this general architecture. Although these systems use TrustZone's hardware based isolation to ensure that applications running inside the secure world are not tempered by a compromised rich OS, they were implemented with the goal of reducing the TCB in order to ensure a less vulnerable system. there are some limitations regarding the applications it can support.

A reduced TCB means that most features of standard mobile operating systems are not supported, for instance, in AndixOS the secure world kernel lacks drivers for peripherals such as the touchscreen or code to control the framebuffer, thus it it is not capable of supporting trusted UI. The same limitations are present in both TLR and OP-TEE. Instead, these sytems support an RPC-like mechanism for in-between-world communication, secure persistence storage and basic cryptographic systems allowing for the development of simple trusted applications.

On-board Credentials (ObC) [18] is another TEE system which supports the development of secure credential and authentication mechanisms. Originally

[12] https://wiki.linaro.org/WorkingGroups/Security/OP-TEE

[13] http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/papers/webcrypto2014_submission_25.pdf

developed for Nokia mobile devices using the TI M-Shield technology, and later ported to ARM's TrustZone, it suffers from the same limitation described above.

2.3.2 TA with Untrusted UI

Trusted Applications are built on top of secure runtimes in order to minimize and mitigate the impact of the normal world on its execution. This subsection describes trusted applications built using runtimes which do not support trusted UI, or TAs which do not require trusted UI to correctly perform its function. Some trusted applications use custom TEEs to fully control the underlying hardware and design specific security solution which may not be supported by generic secure runtimes.

Brasser et al. [4] designed a minimalistic runtime to support the bare minimum necessary for their trusted application to work. This system leverages Android in the normal world and allows for third parties (hosts) to regulate how users (guests) use their devices (e.g., manage their devices' resources), while in a specific space. This system comprises of authentication and communication mechanisms between guest's secure world and hosts, remote memory operations (reads retrieve certain normal world kernel pages and writes change certain page configurations, e.g., uninstall peripheral drivers, either pointing their interfaces to NULL, or by pointing them to dummy drivers, that just return error codes) and other common features supported by most TEE such as secure boot through TrustZone, with secure world component integrity checks, and world switches based on software interrupts.

Other trusted applications are built using generic runtimes and frameworks. DroidVault [23] introduces the notion of data vault as a protective measure for sensitive data on Android mobile devices. To achieve this, DroidVault adopts the memory manager and interrupt handler from Open Virtualization's SierraTEE [14] and is implemented with a data protection manager, encryption library and a port of a lightweight SSL/TLS library called PolarSSL [15]. Much like the trusted application described above, DroidVault supports world switching through software interrupts and secure boot and even inter-world communication. With this TA a user can securely download a sensitive file from an authority and securely store it in the untrusted Android OS, because secure storage space is limited, and allows an app to process sensitive data but only through data protection managers signed by the data owner running inside the secure world. This system supports trusted input/output by using a secure keyboard and serial console, but because we are interested in trusted user interfaces similar to those offered by mobile operating systems we decided to consider this system as not supporting trusted UI.

Android Key Store [16] is another security service in Android. This service allows for cryptographic keys to be stored in a container (keystore), so its extraction from the device becomes difficult and so they can be used for common

[14] http://www.openvirtualization.org/
[15] https://polarssl.org/
[16] http://developer.android.com/training/articles/keystore.html

cryptographic operations. The encryption and decryption of the container is handled by the keystore service, which in turn links with a hardware abstraction layer module called "keymaster". The android open source project (AOSP) provides a sofware implementation of this module called "softkeymaster", but device vendors can offer support for hardware based protected storage when available by using TrustZone.

2.3.3 GPFR with Trusted UI

In this section we describe TEE frameworks and runtimes with support for trusted UI. As referenced before, GlobalPlatform defined standard APIs for the communication between the rich OS running in the normal world and the secure OS, but this organization also defined device specifications which TEEs must comply to in order to be certified. Included in these device specifications is a trusted UI clause, meaning that every TEE which complies with GlobalPlatform's device specifications must support trusted UI. SierraTEE [17], T6 [18] and Open-TEE [27] comply with this standard, and for this reason allow the development of trusted applications with secure user interfaces. Open-TEE's trusted UI feature is being developed by the community as it was not originally supported.

The Genode OS Framework [19] is a tool kit for building highly secure special-purpose operating systems to be executed in TrustZone-enabled processors. Genode implements a framebuffer and input drivers to be used by the secure kernel, thus trusted applications running on top of Genode-based TEEs can offer trusted user interfaces. TrustICE [39] is an isolation framework to provide isolated computing environments (ICEs) on mobile devices. A lot of similar solutions provide isolated computing environments using ARM's TrustZone by allowing code to be executed in the secure world, generally in the form of trusted applications. TrustICE aims at creating ICEs in the normal world domain rather than in the secure world. For this reason, TrustICE's architecture is slightly different from those described before. Figure 5 compares TrustICE's architecture with those of traditional TrustZone TEE's, where trusted applications run inside the secure world domain.

TrustICE works by implementing a trusted domain controler (TDC) in the secure world, which is responsible for suspending the execution of the rich OS as well as other ICE's in the system when another ICE is running, thus supporting CPU isolation for running ICE's. For memory isolation a watermarking mechanism is implemented so the rich OS cannot access secure code running in the normal world memory. In order to isolate I/O devices the secure world blocks all unnecessary external interrupts from arriving at the TDC, thus protecting the TDC from being interrupted by malicious devices, the exception being a minimal set of required interrupts to allow trusted UI.

[17] http://www.openvirtualization.org/
[18] https://www.trustkernel.com/products/tee/t6.html
[19] http://genode.org/

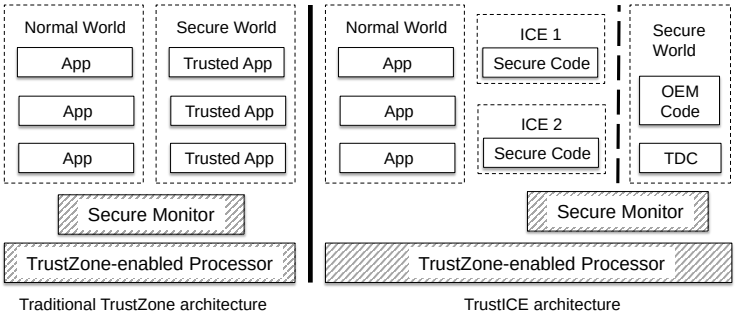

Fig. 5. Architecture comparison between traditional TrustZone's software stack and TrustICE (adapted from [39])

2.3.4 TA with Trusted UI

Most TEEs and trusted applications implement secure touchscreen drivers and secure framebuffers in the secure world such that secure UI can be supported. AdAttester [21] is a mobile ad framework to reliably detect and prevent well-known ad frauds by providing attestable proofs to ad vendors. This is done by providing two novel security primitives: *unforgeable clicks* and *verifiable display*. TrustOTP [38] creates a secure One-Time-Password (OTP) mechanism, secured by hardware, where the OTP is generated based on Time and a Counter secured by TrustZone's memory management. Most trusted applications described before require inter-world communication to trigger the world-switching mechanism. This system leverages hardware interrupts to trigger this world-switch. This mitigates denial-of-service attacks by a malicious rich OS which may control the inter-world communication mechanism. A malicious rich OS may deny calling the necessary methods to trigger the world switch by intercepting the software interrupts, but with this hardware-based solution, even if the rich OS is malicious, the world-switch cannot be avoided by the normal world domain, thus the TrustOTP service is always available even when the rich OS crashes or tries to compromise this service's execution. This solution may be implemented by trusted applications which require no communication or integration with the normal world domain.

TrustDump [37] creates a secure memory acquisition tool and offloads the memory through micro-USB or serial port. Just like TrustOTP, TrustDump has trusted UI and reliable world-switching via hardware interrupts. Samsung KNOX [25] is a defense-grade mobile security platform which provides strong guarantees for the protection of enterprise data. This security is achieved through several layers of data protections which include secure boot, TrustZone-based integrity measurement architecture (TIMA) and Security Enhancements for Android (SEAndroid [36], which was already discussed in previous sections). Samsung KNOX offers a product called KNOX Workspace, which is a container designed to separate, isolate, encrypt, and protect work data from attackers.

This enterprise-ready solution provides management tools and utilities to meet security needs of enterprises large and small. Workspace provides this separate secure environment within the mobile device, complete with its own home screen, launcher, applications, and widgets. The trusted UI for these trusted applications is supported by a secure framebuffer and touchscreen driver implemented in the secure world. This is a viable solution for trusted UI, but implementing a framebuffer and drivers in the secure world highly increases the secure system's TCB, which may introduce code vulnerabilities in the secure domain.

Instead of implementing the required drivers to support trusted UI some systems designed mechanisms to allow the reuse of untrusted drivers implemented in the rich OS by the secure world domain. TrustUI [22] excludes the device drivers for input, display and network from the secure world, and instead reuses the existing drivers from the normal world, thus achieving a much smaller TCB than the systems described above. Because we are only interested in trusted UI, the following explanation discards the network delegation mode. To achieve trusted UI, device drivers are split into two parts: a backend running in the normal world domain and a frontend running in the secure world. Both parts correspond through proxy modules running in both worlds which communicate via shared memory. Whenever secure display is necessary, the frontend asks for a framebuffer from the backend driver and sets the memory region for that buffer as secure only, thus isolating the framebuffer from rich OS manipulation. This mechanism may still be victim of framebuffer overlay attacks, where a malicious backend driver gives a false framebuffer to the secure world. To protect from such attacks the system randomizes the colour of the background and foreground used in the display and uses two LEDs, controlled by the secure world only, to show these same colours. A user can visually check if these colours match with those shown in the display. If they match then the user is assured that the display shown is being controlled by the secure world. To support trusted input from the user TrustUI introduces a randomization mechanism to generate different software keyboards for every input of the user. This means a malicious rich OS cannot correctly intercept user input, but it is still possible to disclose, for instance, the length of a password. For this reason, and to prevent fake key injection attacks, the keyboard mechanism introduces, from time to time, *confirm* buttons randomly position on the screen before continuing.

Summary Table 3 briefly summarizes the categorization of systems described in this section according to the partitioning documented above.

Table 3. TrustZone-based system categorization.

	Untrusted UI	Trusted UI
TAs	Android Key Store DroidVault Brasser et al.	TrustOTP TrustDump AdAttester TrustUI Samsung Knox
GPFR	TLK OP-TEE Andix OS Nokia ObC TLR	Genode T6 TrustICE SierraTEE

3 Architecture

This section describes *TrubiZone*, a novel security framework for TrustZone-enabled devices which supports the development of secure sensitive applications, while leveraging Android as the untrusted rich OS, with the particular purpose of displaying privacy sensitive data to the user. This information may take several formats, either being images, pdfs or straight plain text files. This flexibility allows app developers to easily create several basic secure applications such as a secure password managers, sensitive images display (for instance, for military purpose) and sensitive document managers, without ever being at risk of disclosure by a compromised full-featured operating system. To fully demonstrate the concept of *TrubiZone*, we develop a sensitive mobile health application for managing personal health records. The following subsections describe the underlying *TrubiZone* system architecture and the purposed PHR application.

3.1 TrubiZone

TrubiZone must maintain a small trusted computing base in order to guarantee a secure foundation for sensitive applications. This means only the strictly necessary features for practical applications to work must be supported, and even these features must be built with intelligent and conservative development strategies. Moreover, *TrubiZone* must allow for an integrated environment with Android to provide a transparent isolated security environment for these sensitive applications without compromising the system's usability. To support the described features this project must fulfil the goals of assuring the necessary security policies such as confidentiality, integrity and authenticity of data stored in the untrusted world, must support trusted user interfaces and be developer friendly, which is allow for easy development of simple secure applications such

as those described above. The following paragraphs describe how these goals will be supported in *TrubiZone*.

Secure Storage In order to bypass the physical limitations of storage in the secure world domain a different strategy must be implemented to support secure storage of large files. Inspired by the work which culminated on the development of DroidVault [23], secure storage is possible by enabling large files to be stored in the untrusted domain. To achieve this goal *TrubiZone* must implement, similarly to what is done in DroidVault, a Data Processing Module (DPM), which is the secure module for data operations. A file can be created inside *TrubiZone*, or downloaded from a remote server, and encrypted with a key generated in the secure side, or known by it. This way, sensitive data is encrypted before leaving *TrubiZone* and is never disclosed to the Android OS. Moreover, DPM can verify if a client application running in Android (front-end of a *TrubiZone* trusted application) is the owner of the encrypted data, if it is then the DPM is responsible for loading the corresponding file in the secure world domain so it can be securely managed by the trusted application. The DPM assures the confidentiality and integrity of sensitive data in *TrubiZone* and allows for large files to be securely stored in the untrusted domain of a TrustZone-enabled mobile device.

Secure World-Switching A big limitation of some systems described in section 2 is world-switching based on software interrupts, which allow denial-of-service attacks by a malicious rich operating system that may sabotage inter-world calls responsible for the world switch. To mitigate such attacks, hardware interrupts responsible for the world-switching mechanism were implemented by the authors of TrustOTP [38] and TrustDump [37] in these systems. Non-maskable interrupts (NMI) are hardware interrupts that cannot be ignored by standard interrupt masking techniques, typically used to signal attention for non-recoverable hardware errors, can be used in order to feature secure world switches between the normal and secure world domains. With these interrupts the user can be guaranteed of an inter-world switch and that the operations dependent of this world switch are authentic, this is because the secure domain is non-reentrant, meaning that after the system enters the secure domain, the system will switch back to the rich OS only when *TrubiZone* explicitly triggers the switch.

Trusted User Interface Most trusted execution environments support the execution of code on the secure side, thus supporting some basic secure services. But without a user interface support for user-machine interaction, these systems limit greatly the potential for practical trusted applications. Because user interfaces play a big role in improving app usability, trusted applications must support trusted user interfaces to be useful and so the data from and to the trusted application cannot be eavesdropped by the rich operating system. To achieve this a secure display controller must be implemented to securely copy the image from a secure framebuffer to the display device, where the framebuffer

stores the image to be displayed. By having different framebuffers for the secure and normal domains, *TrubiZone* can prevent potential data leakage, as the secure framebuffer is reserved for use in the secure world. Moreover, because usually resources such as the video card and display screen are shared by both domains, the reliable display controller must be able to correctly program both resources. On the other hand, to support trusted input from the user, a self-contained secure screen driver must be included in the secure world domain. With these mechanism there is a guarantee that the information displayed to the user and the information given as input to the system is authentic.

Developer Friendly Trusted execution environments already present some developer friendly strategies to easily allow new trusted application development. This is generally done by supporting applications built using GlobalPlatform's APIs. But to feature more complex mechanisms, such as trusted user interfaces, some trusted applications have to manage low level resources, such as the graphical framebuffer, directly. This highly increases the complexity of developing trusted applications and is unnecessary to implement specifically for every app as most of them require such mechanims to be useful. For this reason, *TrubiZone* implements an API for an easy development of trusted applications which leverage resources such as the graphical framebuffer (for trusted UI), secure shared memory and inter-world communication, and secure storage.

Architecture

3.2 Personal Health Records Application

4 Implementation and Evaluation

Implementation: TrubiZone will run along side Android and will be implemented upon a Freescale i.MX53 Quick Start Board development board [20], which supports ARM's TrustZone. To illustrate *TrubiZone*'s functionality a Personal Health Record management mobile application will be implemented. This applications allows a user to display, store and edit his personal health information securely, without this sensitive information ever being exposed to a untrusted rich OS such as Android. To allow for a more flexible development environment using TrubiZone, we expect to support sensitive applications built using GlobalPlatform's API.

Evaluation: To evaluate *TrubiZone* we measure the performance overhead for displaying, storing and editing sensitive files as well as compare the development process for the PHR mHealth app using TrubiZone and a similar app built on top of Android. The performance can be measured by using the performance monitor available in the Cortex-A8 processor to count the CPU cycles and then convert

[20] http://www.freescale.com/products/arm-processors/i.mx-applications-processors-based-on-arm-cores/i.mx53-processors/i.mx53-quick-start-board:IMX53QSB

the cycles to time by multiplying 1 *ns / cycle*. By conducting each experiment for each of the use cases described several times and averaging the value we can compare this value taken for *TrubiZone* with the value measured in the same conditions for a similar applications without using the secure system.

5 Future Work Plan

Future work is scheduled as follows:

- January 9 - March 25: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 26 - May 1: Perform the complete experimental evaluation of the results.
- May 2 - May 10: Write a paper describing the project.
- May 11 - June 15: Finish the writing of the dissertation.
- June 15: Deliver the MSc dissertation.

6 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. Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating smart personal devices in restricted spaces.
5. 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.
6. 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.
7. 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.

8. Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Information Security*, pages 331–345. Springer, 2011.
9. Miguel Costa. SecMos - Mobile Operating System Security. Master's thesis, Instituto Superior Técnico, Portugal, 2015.
10. 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.
11. OMA DRM. Open mobile alliance digital rights management.(2010). *Retrieved May*, 2, 2011.
12. Nuno Duarte. On The Effectiveness of Trust Leases in Securing Mobile Applications. Master's thesis, Instituto Superior Técnico, Portugal, 2015.
13. 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.
14. Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. The andix research os-arm trustzone meets industrial control systems security.
15. 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.
16. 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.
17. 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.
18. Kari Kostiainen et al. On-board credentials: an open credential platform for mobile devices. 2012.
19. David Kotz. A threat taxonomy for mhealth privacy. In *COMSNETS*, pages 1–6, 2011.
20. 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.
21. Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. Adattester: Secure online mobile advertisement attestation using trustzone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 75–88. ACM, 2015.
22. Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.
23. Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Pratiksha Saxena. Droidvault: A trusted data vault for android devices. In *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*, pages 29–38. IEEE, 2014.
24. 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.

25. Samsung Electronics Co. Ltd. Samsung KNOX - White Paper : An Overview of Samsung KNOX. Technical report, Samsung Electronics Co. Ltd., 04 2013. http://www.samsung.com/es/business-images/resource/white-paper/2014/02/Samsung_KNOX_whitepaper-0.pdf.
26. 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/.
27. Brian McGillion, Tanel Dettenborn, Thomas Nyman, and N. Asokan. Open-TEE – an open virtual trusted execution environment. Technical report, Aalto University, 2015.
28. 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.
29. 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.
30. 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.
31. 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.
32. Research2Guidance. Mobile Health Market Report 2013-2017. Technical report, Research2Guidance, 03 2013. http://research2guidance.com/.
33. 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.
34. Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Trusted language runtime (tlr): enabling trusted applications on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 21–26. ACM, 2011.
35. Bilal Shebaro, Oluwatosin Ogunwuyi, Daniele Midi, and Elisa Bertino. Identidroid: Android can finally wear its anonymous suit. 2014.
36. Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.
37. He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Reliable and trustworthy memory acquisition on smartphones. *Information Forensics and Security, IEEE Transactions on*, 10(12):2547–2561, 2015.
38. He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 976–988. ACM, 2015.
39. He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 367–378. IEEE, 2015.

40. 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.
41. 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.
42. 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.
43. 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.