ViewZone

Embedding Secure Displays in Insecure Mobile Interfaces 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. Mobile devices are growing and being adopted everywhere. Increasingly, such devices execute critical applications which require very sensitive data to be displayed on the devices' screen. Medical and financial records are examples of such sensitive data. However, in order to display sensitive information, applications currently depend on the code of all underlying layers, which comprise a full-blown operating system. Given that all these software layers result in a very large and complex computing base, it is difficult to ensure that the sensitive records displayed on screen have not been intercepted and leaked by a malicious agent that managed to compromise the operating system. The goal of this thesis is to design and implement a TrustZone-based solution that enables applications to securely output sensitive data without relying on a large rich OS. This report describes the state of the art and introduces a preliminary architecture of our solution: a software system named *ViewZone*. To motivate and validate the concept of this solution, a critical mobile health application will be implemented.

1 Introduction

Mobile devices are becoming the predominant platform for simple everyday computing activities. A recent study [9] shows that 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. This shift is explained by the fact that actions that required powerful desktop computers can now be easily performed on mobile devices.

Along with the proliferation of mobile devices, mobile applications (apps) have started to handle privacy and security-sensitive data, such as photos, health records, banking information, location data and general documentation. Mobile health (mHealth) applications, for example, handle medical data, which according to Reuters [37] is more valuable than credit card information. In 2013, Research2Guidance [36] 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. This market is expected to grow even further, from a $6.21 billion revenue in 2013 to $23.49 billion by 2018 [29].

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. Data becomes particularly vulnerable when encryption cannot be used, namely when such data must be displayed on the device's screen. For example, when browsing through sensitive health records, the data must be written in the framebuffer in its unencrypted raw form so that it is readable by the user. Thus, in a malware-infected operating system, data can easily fall prey to an attacker who can steal it from the framebuffer and leak it out.

Currently, providing secure output capability for commodity mobile platforms, such as Android or iOS, is very challenging. Essentially, to protect sensitive data, mobile app developers rely on mechanisms implemented by the operating system. Such mechanisms include access control [42,22,10,21], application communication monitoring [32,12,7] and privacy enhancement systems [5,50,40,15], either from native Android, iOS and Windows or from extensions. But these mechanisms do not offer support for trusted user interfaces, i.e., to securely display sensitive information. Instead, 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.

To reduce the TCB and offer an isolated environment for security critical applications, the research community has developed trusted hardware technologies [27], trusted execution environments [18,28,30,41] (based on microkernels), and trusted services [26,6,25,44]. But these systems generally do not support or make it very hard for application developers to offer secure display seamlessly integrated into the application's user interface.

1.1 Goals

The goal of this project is to fill in a security gap in the mobile application market by proposing *ViewZone*, a software system that provides trusted display capability for Android applications. Such a capability will enable applications to render encrypted sensitive content, such as images and text, without the need to trust the code of both the application and the operating system. Decrypting and rendering the content will be performed by the *ViewZone* software living in an execution environment isolated from the operating system. To enforce isolation, *ViewZone* will leverage the security primitives of ARM TrustZone, a trusted hardware technology widely available on commodity mobile devices. This ensures that the data displayed by the device is not intercepted by a malware-infected operating system. Thus, the central goal of this thesis is the design and implementation of *ViewZone*.

While providing a trusted display service, *ViewZone* must fulfil the following additional requirements:

Small Trusted Computing Base
To ensure the security of *ViewZone* and of the trusted services running on top of it, this implementation must have a small TCB, comprising of a few KLOCs. This can be done by using TrustZone to isolate the secure service responsible for the trusted display from the rich operating system and by designing a carefully crafted trusted kernel responsible for rendering the protected content on screen.

Support for General Applications
The implementation should allow generic Android applications to securely display sensitive content without relying on operating system mechanisms, which may be compromised. Additionally, the system should support the coexistence of trusted and untrusted output, thus allowing the user not to be aware of a context change between the untrusted generic application and its trusted counterpart.

Developer Friendly
Developers must be able to easily specify what needs to be displayed in the trusted environment. This can be done by supporting simple and familiar programming abstractions to render protected content in their apps. This way the system can be used as a secure display service for Android applications.

In summary this work expects to contribute with:

- The design of a TrustZone-based software system that provides secure output channels for Android applications while depending on a small TCB;
- Implementation, on a development board, of the system's prototype;
- Implementation of a mHealth application using the prototyped system;
- Experimental evaluation of the prototype;
- Evaluation of the mobile health app as a system use case.

The remainder of this document proceeds as follows. Section 2 discusses the related work. Section 3 introduces the architecture of *ViewZone*. Section 4 highlights the evaluation methodology and implementation. Section 5 describes the work plan for this project. Section 6 concludes the report.

2 Related Work

This section describes the related work and is organized in three main parts. First, in order to better motivate our proposal, we discuss the existing security issues of a class of applications that can take advantage of a trusted output functionality: mobile health applications. In the second part of this section, we overview the state of the art of general-purpose mobile security mechanisms used to implement most secure sensitive applications, such as those described in the first part. We also explain why such mechanisms fall short in supporting secure output to these apps. Finally, the third part describes a specific class of security systems which, similarly to *ViewZone*, leverage TrustZone to implement a variety of security solutions for mobile platforms.

2.1 Studies in Mobile Health Security

Medical data is highly profitable for malicious agents who can use it for medical identity theft, as it can be more valuable than credit card information. Generally, vulnerabilities which lead to sensitive data leakage stem from negligent development of healthcare systems. For this reason, regulatory laws such as the Health Insurance Portability and Accountability Act (HIPAA) have been established. These laws comprise the standard for electronic healthcare transactions and must be followed by all developers when managing sensitive health data.

In the mobile context, data is even more exposed and vulnerable due to several reasons: *(1)* inherent portability of these devices, *(2)* sharing of information to third-party advertisers by device manufacturers or mobile app developers, *(3)* unregulated management of sensitive medical information, specially because regulatory laws such as HIPAA do not account for the mobile sector, and *(4)* existence of security flaws in consumer device software, which can be exploited by malware. Such an exposure to attacks motivates the need for a security system capable of allowing applications to securely display and manage sensitive data.

The following subsections describe work done in assessing the security properties of commercial mobile health applications as to better understand the need for a secure content display capability. The first subsection describes a threat taxonomy for mHealth applications by Kotz [24]. The second and third subsections describe work done by He et al. [20] through the analysis of three studies which answer the following questions: *what are the potential attack surfaces* and *how widespread and how serious are these security threats.*

2.1.1 Threat Taxonomy for Mobile Health Applications

Kotz [24] 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 misuse 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 medical identity theft. And *outsiders* may be able to observe patient identity or location from communications.

Access threats are described as unauthorized accesses to PHR and include scenarios where the patient controlling the data allows broader-than-intended access or disclosure of information. Additionally, *insiders* may also snoop or modify patient data with malicious intent, and *outsiders* may leak or modify this data by breaking into patient records.

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 [49]. The

Table 1. Description of attack surfaces (taken from He et al. [20]).

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 in unencrypted files on the 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 Channels	Sensitive information can be inferred by a malicious app with side channels, e.g. network package size, sequence, timing, etc..

adversary may also use physical or link layer fingerprinting methods to identify the device type and inject or selectively interfere with wireless frames.

Avancha et al. [3] developed a privacy framework for sensitive mobile health systems. A developer should use the list of privacy properties provided by this article as a check-list to be considered in any design. But even these security properties are not sufficient to withstand an attack leveraged by a compromised operating system, which is an attack not explicitly described in the above taxonomy. For this reason, there is a need for a security solution capable of withstanding such attacks whilst protecting sensitive information.

2.1.2 Potential Attack Surface of Commercial Mobile Health Applications

He et al. [20] analysed several mHealth applications available in Google Play (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 are determined to be in need of protection. These attack surfaces are shown in Table 1. This table provides an overview of the problems disclosed by the apps studied. From the table, we can observe that applications send unencrypted information or use incorrectly configured Internet protocols. These apps also store data in third-party servers, where data is not assured to be secure. Additionally, these apps inadvertently log sensitive data to the system or export private app components, making them accessible to other applications.

However, Table 1 is missing an important attack surface, which is the OS. Operating systems have a large and complex code base, thus increasing the incidence of coding vulnerabilities. These vulnerabilities may be explored by malware which can then control system resources and the overlying applications.

Resources, such as the framebuffers or touchscreen device, might be explored by a malicious agent in order to disclose data, such as sensitive medical information.

He et al. also document that most apps targeted for patients (60%) are in the Life Management category, which comprises nutrition and fitness apps. These are followed by apps that manage and synchronize user health information (PHR Management), which occupy nearly half (46.88%) of the domain. These numbers are a good indicator of the data handled by most commercial mHealth apps available, as well as which health apps are more used in the mobile health sector. From these numbers we can infer that a personal health record viewer can serve as a representative use case to demonstrate a trusted output solution.

2.1.3 Severity of Attacks to Mobile Health Applications

In the second study by He et al. [20], 27 of the top 1080 free apps from the Medical and Health & Fitness categories on Google Play were analysed in order to assess the most commonly observed vulnerabilities. From this analysis, three attack surfaces are identified as the most important ones: *Internet*, *Third Party Services* and *Logging* (see Table 1). Only 7 of the 27 apps use the Internet to effectively send medical information over to remote servers. Some of these apps send unencrypted content, which generally include emails, usernames and passwords. This study also concludes that most 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. This is mostly because users are not capable of assessing whether the data is encrypted, and thereby preventing hosting companies from accessing it.

In the third study by He et al. [20], another 22 apps, which send information over the Internet, are randomly selected from the same top 1080 apps. These apps were then audited to understand what information is effectively being sent, 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. This data includes personal profiles, health sensor data, lifestyle data, medical browsing history and third-party app data (e.g., Facebook account information).

The consequences of disclosing or tampering sensitive health data depend on the type, sensitivity and volume of data breached. However, it is clear that profiling, medical identity theft and healthcare decision-making errors are all possible. This is why He et al. suggest the use of encryption for communication and storage, and encourage developers to create a set of standard privacy guidelines that offer a baseline for protection. Nonetheless, these measures are not sufficient to protect sensitive data against more complex attacks, such as those which involve a malware compromised OS. In a system where a malicious entity controls system resources, encryption techniques and good developing guidelines might not be enough. This is because the system can bypass these mechanisms and disclose data when it is in its raw displayable form, i.e., unencrypted.

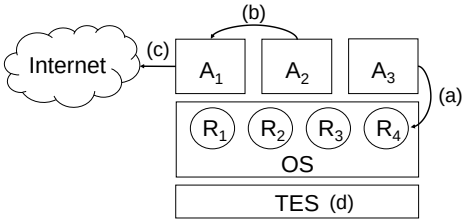

Figure 1. Security mechanisms map; A-application, R-system resource, TES-Trusted Execution System.

Security measures and regulatory laws are implemented for custom healthcare information systems in order to mitigate attacks which may disclose sensitive medical data. However, in the mobile sector, these security measures may not be possible or practical to implement, and regulatory laws, such as HIPAA, do not account for the mobile market. For this reason, the research community analysed and categorized mHealth threats, developed a privacy framework and studied commercially available apps in order to achieve a standardized set of rules to mitigate negligent development. But these studies consider the whole operating system as the trusted computing base for mHealth applications. Because this TCB is so complex and unreliable, malware may take control of system resources and consequently leak sensitive data. Ultimately, what the mHealth market needs is a security solution which provides secure primitives that both developers and users desire, such as secure display. And this secure display must work without having to rely on a full-featured operating system, inherently with a large TCB.

2.2 General-Purpose Mobile Security Mechanisms

Like most existing mobile applications, mHealth apps tend to depend on the security services provided by the mobile platform. This section provides an overview of the security mechanisms that have been developed for Android.

Since many security mechanisms have been proposed, we divide them into groups to reflect the different ways in which sensitive information is protected: *(a) access control mechanisms*, which enforce security policies that prevent access to sensitive resources by a given application, *(b) application communication monitoring*, which ensures that sensitive data that was read by a given application can be securely exchanged with co-located apps, *(c) privacy enhancement systems*, which aim at preventing privacy breaches by applications that read sensitive data, and need to share it with Internet services, and *(d) trusted execution systems* (TES), which provide restricted environments in which sensitive data can be processed while trusting in a smaller code base than that of a rich OS. Figure 1 maps these four groups to a representation of a possible mobile system.

2.2.1 Access Control Mechanisms

Access control mechanisms implement security models in which subjects (e.g. user, processes, threads, etc.) are constrained by a security policy to perform certain actions on the system, namely accesses to resources, typically called objects (e.g. files, sensors, etc.).

Android inherits a Discretionary Access Control (DAC) mechanism from its Linux based kernel, but some system resources, such as the IPC Binder mechanism, are accessed via Mandatory Access Control (MAC) policies. In a DAC system the data owner is responsible for the data and, as such, determines who can access it. In Android, an application can create and store files in the filesystem, thus becoming the sole owner of such files, and it can allow access to them for any other application. In MAC, subjects are much more restricted in determining specific permissions because the restrictions on these resources are defined by a global system policy. In the Android operating system, once a subject attempts to access an object, it triggers a policy evaluation by the kernel, which assesses whether the access is granted. The advantage of this strict system is its robustness and restrictiveness, 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 more restrictive, several systems were created over the years to extend MAC's access control model to other Android resources. SEAndroid [42] solves problems related to resources complying with the DAC mechanism. The authors ported SELinux [34] to provide MAC at the kernel layer. The kernel was then modified to support a new MAC policy (e.g., filesystem, IPC). A new middleware layer (MMAC) was also created to extend MAC to Android's Binder IPC. TrustDroid [8] extends the MAC mechanism to all the platform's resources in order to isolate different domains' sensitive information.

Permission Refinement In spite of its robustness, the Android permission mechanism is a very restrictive. At install time, the system displays on screen the list of application-specific permissions that must be accepted or denied by the user. The user is forced into a binary decision, either granting all permissions or quitting the installation. Furthermore, some of these applications may even request more permissions than those effectively needed. This is an inflexible solution, which makes it impossible for users to have full control over the permissions an application effectively requires at runtime. This inflexibility allows an app to use the device's resources without the user's knowledge, possibly with malicious intent.

Over the years, many systems aimed at improving the state of affairs through permission refinement. APEX [31] modified Android's permission model to allow users to specify the assignment of permissions to each app, both at install and runtime. Permission Tracker [22] allows users to be informed on how permissions are used at runtime and offers the possibility of revoking those 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 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 automatically restrict permissions to all applications running along side a security sensitive app, thus isolating this application and avoiding possible leaks by shared resources.

There are several context-aware permission refinement systems developed by the research community. Trusted third-parties can use CRePE [10] to enforce security policies on other devices. For example, an employer may enforce a security policy on the employees' mobile devices when inside the company. Similarly, MOSES [38] enforces domain isolation through the concept of security profiles, allowing it to switch profiles based on pre-established conditions (e.g., GPS coordinates and time). Additionally, MOSES leverages TaintDroid [15] 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 Most security extensions, such as CRePE [10] or MOSES [38], require modifying and adding components to the kernel and middleware layers in order to implement new security models. Some frameworks, such as Android Security Modules (ASM) [21] and Android Security Framework (ASF) [4], were built in order to ease this development process. These frameworks allow developers to easily create new security models as ordinary Android applications whilst benefiting from a full callback system, which allow apps to be notified of accesses to resources of interest.

These frameworks comprise a set of hooks distributed along 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 hook activation may or may not proceed. The main advantage of these frameworks is the flexibility and freedom given to developers in choosing whatever resources to manage.

Memory Instrumentation An alternative approach to implementing access control policies is based on memory instrumentation. Memory instrumentation leverages application code analysis techniques to restrict access from those applications to the corresponding resources. Memory instrumentation can be divided into two groups: *static memory instrumentation* and *dynamic memory instrumentation*. While static memory instrumentation changes pre-compiled bytecode, dynamic instrumentation patches running processes, and for this reason it supports the enforcement of new security models. DeepDroid [48] relies on dynamic memory instrumentation to enforce fine-grained context-aware security policies for enterprise and does this by patching several system services and tracing system calls to resources of interest.

Digital Rights Management is a specific access control technology which allows data owners to restrict if and how their data can be copied and also how it can be 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.

In order to define the format of the content delivered to DRM Agents and how this content can be transferred from the Content Issuer to the DRM Agent, the Open Mobile Alliance (OMA) developed the DRM standard [13]. Android provides an extensible DRM framework, called Android DRM Framework [1]. This framework allows application developers to enable their apps to manage rights-protected content by complying with one of the supported DRM schemes (mechanisms, enforced by DRM Agents, to handle particular types of files).

To understand how DRM could be employed in the context of mHealth, one can suggest the 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. These restrictions are imposed upon the content, i.e., the personal health record, 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*. Although this solution seems to fit the security properties a critical application requires, once again the whole operating system belongs to the trusted computing base. Therefore, resources shared by different applications such as the framebuffers and display devices may leak sensitive information if the system becomes compromised.

2.2.2 Application Communication Monitoring

Along with access control enforcement, some concerns have been raised about the security of Android's standard Inter-Process Communication (IPC) mechanism. This mechanism allows different processes, i.e., applications, to communicate with each other, and controls how apps access system components. However, this mechanism can be abused by applications, as they can leverage it to access unauthorized resources or data. In this section we describe two attacks on Android's IPC mechanism followed by systems developed to mitigate such attacks. The attacks described are called *Confused Deputy Attacks* and *Collusion Attacks* and are represented in Figure 2.

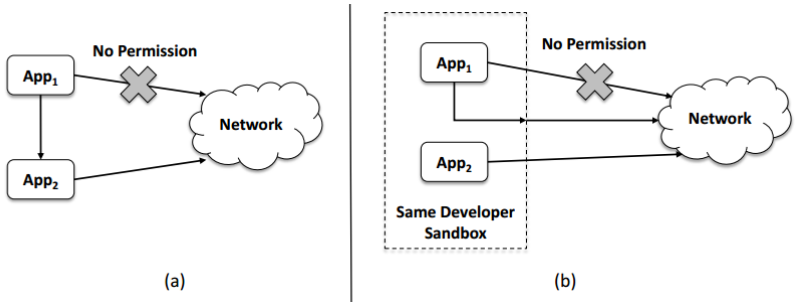

Figure 2. Confused deputy attack (a); Collusion attack (b) - (taken from Duarte [14]).

Confused deputy attacks basically consist of unprivileged applications taking advantage of other applications' publicly accessible APIs to perform an unauthorized action. If App 2 publicly allows other apps to use the Internet through its API, then the user is unaware that App 1 can use the Internet without explicitly requesting it in its manifest. Collusion attacks consist of an app that although not granted the permission to perform an action, can nevertheless perform it. This is possible if there exists another app, belonging to the same developer and 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. To mitigate such attacks the research community developed Saint [32] and QUIRE [12]. Saint was created to specify which apps can access the public APIs of another app. QUIRE denies access to an API if the message exchanged between apps stems from an unauthorized source. QUIRE does this by analysing the full call chain context of the message, which contains the source of the request.

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 [7] 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 populating a graph at runtime with a representation of the apps' interactions, which is then used to regulate inter-app communication.

Although these systems can be used to avoid promiscuous communication between applications, they do not support the development of apps with the goal of displaying sensitive content securely.

2.2.3 Privacy Enhancement Systems

The mechanisms reviewed so far cover the security issues involved when an application *1)* reads data from a given resource, or *2)* attempts to share sensitive

data with co-located applications. In addition, applications may try to exfiltrate sensitive data to remote untrusted parties, which raises privacy concerns. In fact, the biggest concern for privacy-sensitive mobile applications is data leakage, specially with valuable data such as health records. In this section we describe systems which aim to control the flow of sensitive data in order to assess whether such data leaves the systems as a consequence of a malicious action.

Systems like MockDroid [5], TISSA [50] and IdentiDroid [40] are extensions to Android's data access control mechanism and prevent untrusted applications from accessing sensitive data. This is done by allowing users to manually specify application access rights over the system services, such as geographic location. These systems may also provide data shadowing[1] to return plausible but incorrect results to the requesting application.

Alternative systems use a different approach to solve the sensitive data disclosure problem. Unlike data access control mechanisms, dynamic taint analysis systems, such as TaintDroid [15], 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 user is then 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.

Although these systems can be leveraged in order to understand if sensitive data is leaving the system through network sinks, they fall short in assessing whether data is accessed via shared resources, such as framebuffers and display devices. This means that a compromised OS may access such resources and disclose sensitive information.

2.2.4 Trusted Execution Systems

In all previously discussed systems, the mobile applications depend entirely on the integrity and correctness of the operating system. In other words, the trusted computing base includes the full-featured operating system responsible for the app's execution. To reduce the trusted computing base, *trusted execution systems* (see Figure 1) allow certain (in some cases all) operating system components to be disabled or deemed untrusted and providing a restricted execution environment in which sensitive application code and data can be executed safely.

TrUbi is a system, developed by Duarte [14] and Costa [11], built on top of ASM [21] which allows for flexible system-wide resource restriction. This system 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, a simple PHR app to display personal health records could be

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

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 (but the key, because it is generated from a user password, is not stored on the system).

Although TrUbi effectively reduces the TCB dynamically by disabling concurrent running processes, it still trusts the operating system's code base, which may be compromised by malware. Furthermore, TrUbi has no support for trusted UI meaning that a compromised OS can access sensitive data by disclosing the content of resources such as graphical framebuffers and display devices. The following systems allow for a more significant reduction of the TCB by isolating the execution environment of sensitive code from that of a full-featured OS.

External and Internal hardware security modules External hardware security modules represent the classic security solution for embedded applications, which consists of offloading trust from the OS into a dedicated piece of hardware commonly named *trusted element* (e.g., a smartcard). Typically, the trusted element is located outside of the main Socket-on-Chip (SoC). 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. Internal hardware security modules are, contrary to external modules, 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. This processing engine 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 modules.

However, the main disadvantage of both external and internal security modules is that they only provide 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. In both of these hardware-based security solution, sensitive information from the trusted element must always go through the rich OS before it can be written to the framebuffer, opening the window for interception by malicious agents. The next system solves this problem by allowing complete isolation between different execution environments.

TrustZone is a hardware architecture designed to allow the execution of code in isolation from the rich operating system (see Figure 3). In this architecture, the full-featured operating system runs in the normal world domain, while the trusted code running in the secure world can execute without relying on a complex code base. TrustZone also mitigates performance overheads inherent to software virtualization techniques since the hypervisor mechanism, which manages the trusted and untrusted domains, is implemented natively in hardware.

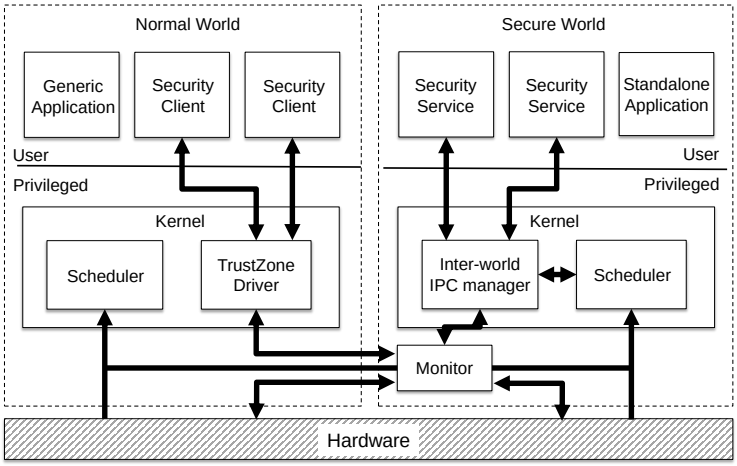

Figure 3. TrustZone's Software Architecture (adapted from [27]).

TrustZone hardware and software architecture are described in a whitepaper [27] 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 cores 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 3.

This design allows for concurrent execution of multiple secure world apps and services that are completely independent from the normal world environment, thus, even with a compromised normal world, the secure world executes as expected. Moreover, 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. These advantages sparked the research community in developing several systems which leverage this technology. For this reason, TrustZone will be discussed in depth in Section 2.3.

2.2.5 Summary

After describing the state of the art of general-purpose mobile security mechanisms, we discuss to what extent existing systems have the potential to provide trusted output paths.

Access control mechanisms focus on restricting access to system resources, which may be useful for critical applications if these can use it to limit the access to shared resources by less sensitive applications running in parallel. But these mechanisms rely on a full-blown operating system, with a large and complex TCB, to ensure this property. And this full-blown OS may become compromised

by malware and sabotage these security measures. Moreover, these systems do not offer trusted user interfaces as they provide no explicit isolation between the operating system and resources such as display devices and framebuffers. For these reasons, access control mechanisms do not fulfil the needs of this project.

Application communication monitoring focuses on assuring that communication attacks cannot effectively target Android's IPC mechanism, which is the main communication component between applications in the Android platform. Because these systems are not meant to be used for the development of applications, but rather as a protective measure against specific attacks, these mechanisms are not suitable to solve the underlying problem of developing critical applications to securely display sensitive data.

Privacy enhancement systems focus on assessing whether sensitive data leaves the system via network sinks. While these systems are useful in controlling the data flow of traditional paths, such as Android's IPC, it falls short of controlling the data flow of unconventional paths, such as framebuffers and display devices. This means that sensitive data may be intercepted by a malicious OS without the privacy enhancement system knowing it.

Trusted execution systems focus on supporting an isolated environment which comprises a smaller TCB than that of a full-featured OS. This category comprises very different systems, with different purposes and approaches, which we summarize next. TrUbi dynamically reduces the overall system's TCB while a critical application is running. But this is done whilst relying on the operating system, which may be compromised.

External and internal hardware security modules offer an isolated hardware-based environment for the execution of secure tasks. But because both external and internal modules provide only secure processing and storage functions, the data it protects must eventually be used outside the isolated environment, for instance for display purposes. This means that these systems still rely on the untrusted OS for functionality such as trusted UI, which we want to be supported by the secure environment.

Lastly, TrustZone also supports isolated environments running along side one another by leveraging special purpose hardware to isolate both domains, which mitigates the performance overhead of software based isolation. Additionally, TrustZone also supports trusted user interfaces by controlling the necessary peripherals. For this reason, TrustZone will be discussed in depth in the next section, as it supports all the features necessary to implement *ViewZone*.

2.3 TrustZone-based Mobile Security Systems

To improve security, mobile device manufacturers have been designing hardware architectures enhanced with *trusted hardware*. Among the available security architectures there is ARM's TrustZone technology, a trusted hardware which allows the development of a diverse set of security systems and services, such as Samsung KNOX [28] and DroidVault [26]. TrustZone is becoming popular as it supports code to be executed isolated from a full-featured operating system

such as Android. This enables a reduction of the trusted computing base of which critical applications depend.

One of the most important uses of TrustZone is building Trusted Execution Environments (TEE), which are compact systems running in the secure world to provide an isolated environment for critical applications. Since its formal standardization by the OMTP in 2007, several Trusted Execution Environment (TEE) software stack architectures have been implemented. This standard comprises a set of security requirements on functionality a TEE should support. The GlobalPlatform [19] 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.

Table 2 categorizes the existing systems in two main dimensions. When surveying the literature, TrustZone-based systems can be divided into two separate groups depending on whether they support general or specific application code hosting: Trusted Kernels, and Trusted Services. Trusted Kernels, which comprise the TEEs, allow the execution of generic code in the secure world environment, whilst a Trusted Service implements a special-purpose application in the secure domain and can run directly on bare metal (e.g., a secure key store, an authentication service, etc.). Orthogonally, one can classify both groups with regards to support for trusted user interface (UI). A TrustZone-based system features Trusted UI if it allows secure world components to directly access the mobile interface without interference from the rich OS, thus minimizing the risk of, for instance, password logging. Since *ViewZone* aims to offer a specific functionality for enabling secure display of application data, our system can be considered as a Trusted Service with trusted UI, which fits the second quadrant of Table 2.

The remainder of this section describes in more detail the existing TrustZone enabled systems according to all four categories: *(i)* Trusted Kernels with Untrusted UI, *(ii)* Trusted Services with Untrusted UI, *(iii)* Trusted Kernels with Trusted UI and *(iv)* Trusted Services with Trusted UI.

2.3.1 Trusted Kernels with Untrusted UI

Trusted Kernels have the goal of executing generic code in its isolated environment, and most of theses kernels have similar architectures (similar to the one described in section 2.2.4). This architecture is generally composed of a small trusted kernel running in the 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 [33], TLK [47], TLR [39] and AndixOS [16] are TEE implementations which share this general architecture. On-board Credentials (ObC) [23] is another TEE system, originally developed for Nokia mobile devices using the TI M-Shield technology and later ported to ARM's TrustZone, which supports the development of secure credential and authentication mechanisms. Although these systems use TrustZone hardware based isolation to ensure that applications running inside the secure world are not modified by a compromised rich

Table 2. TrustZone-based system categorization.

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

OS, they were implemented with the goal of reducing the TCB in order to ensure a less vulnerable system, and for this reason there are some limitations regarding the features they can support.

A reduced TCB means that most features of standard mobile operating systems are not supported. For instance, in both TLR and OP-TEE, as well as AndixOS, the secure world kernel lacks drivers for peripherals such as the touchscreen or code to control the framebuffer, thus it is not capable of supporting trusted UI. For this reason, these systems do not allow developers to easily build trusted applications for sensitive data display. Instead, these sytems support an RPC-like mechanism for in-between-world communication, secure persistent storage and basic cryptographic systems allowing for the development of simple trusted services.

2.3.2 Trusted Services with Untrusted UI

As opposed to Trusted Kernels, which enable the execution of general-purpose application code on the secure world, Trusted Services are designed to implement specific applications in the secure world natively. Some trusted services, such as DroidVault [26] and Restricted Spaces [6], use custom trusted kernels to fully control the underlying hardware and execution environment, whilst designing specific security solutions which may not be supported by generic trusted kernels.

A system by Brasser et al. [6], which will be referred to as Restricted Spaces for the remainder of this section, allows for third parties (hosts) to regulate how users (guests) use their devices (e.g., manage device resources), while in a specific space. This system comprises authentication and communication mechanisms between the guest's secure world and the host's. It also supports remote memory operations, which allow for configuration changes such as uninstall peripheral drivers. This can be done by either pointing their interfaces to NULL, or by pointing them to dummy drivers that just return error codes. With this, Remote Spaces is capable of securely refine permissions using a context-aware approach.

DroidVault [26] introduces the notion of data vault, which is an isolated data protection manager running in the trusted domain for secure file management for Android mobile devices. To achieve this, DroidVault adopts the memory manager and interrupt handler from Open Virtualization's SierraTEE [41] and is implemented with a data protection manager, encryption library and a port of a lightweight SSL/TLS library called PolarSSL [35]. Much like Restricted Spaces, DroidVault supports world switching through software interrupts, secure boot and even inter-world communication. With this Trusted Service a user can download a sensitive file from an authority and securely store it on the device. The sensitive file is encrypted and signed by the data protection manager before it is stored in the untrusted Android OS, in order to save space in the limited storage capacity available at the secure world domain.

Android Key Store [2] 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 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.

The main drawback of the Trusted Services just mentioned is that they do not fulfil the goals of this project as they do not support necessary features such as secure display. On the other hand, DroidVault takes an interesting approach to secure storage support for Trusted Services.

2.3.3 Trusted Kernels with Trusted UI

In this section we describe Trusted Kernels, which allow the execution of generic code in the secure world environment, with support for trusted UI. As referenced in Section 2.3, GlobalPlatform defined standard APIs for the communication between the rich OS running in the normal world and the secure OS. However, this organization also defined device specifications that TEEs must comply in order to be certified. Included in these device specifications there is a trusted UI clause, meaning that every TEE which complies with GlobalPlatform's device specifications must support trusted UI.

SierraTEE [41], T6 [46] and Open-TEE [30] comply with the GlobalPlatform 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 [18] 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.

Samsung KNOX [28] is a defense-grade mobile security platform which provides strong guarantees for the protection of enterprise data. Security is achieved

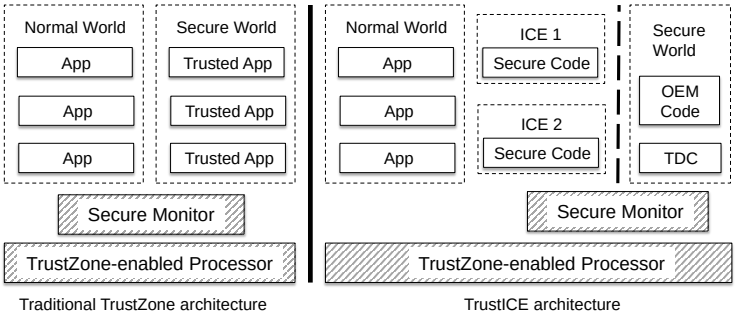

Figure 4. Architecture comparison between traditional TrustZone's software stack and TrustICE (adapted from [45]).

through several layers of data protections which include secure boot, TrustZone-based integrity measurement architecture (TIMA) and Security Enhancements for Android (SEAndroid [42], which was already discussed in Section 2.2.1). 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.

Unlike the solutions previously described, such as Genode OS and Samsung KNOX, which provide isolated computing environments by allowing code to be executed in the secure world, generally in the form of trusted applications, TrustICE [45] aims at creating Isolated Computing Environments (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 4 compares TrustICE's architecture with that of a traditional TrustZone TEE, 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.

Because these systems support secure display they are adequate for the development of trusted applications which require sensitive information to be displayed to the user. However, developing such applications is complex because the application development environments are cumbersome and error-prone.

2.3.4 Trusted Services with Trusted UI

Besides Trusted Kernels, some Trusted Services offer secure display to applications implemented in the secure domain. The technical challenge is implementing drivers in the secure world without greatly increasing the TCB, and this is solved by implementing very small special-purpose display drivers. Several Trusted Services with trusted UI support have been proposed in the literature.

TrustOTP [44] is a One-Time-Password (OTP) system, 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 the world-switch. This mitigates denial-of-service attacks by a malicious rich OS which may control the inter-world communication mechanism and intercept the calls (software interrupts) required to trigger the world-switch.

Providing a different service, TrustDump [43] is a secure memory acquisition tool that offloads the memory through micro-USB. Similarly to TrustOTP, this system relies on hardware interrupts to trigger world-switches. This solution may be implemented by systems which require no inter-world communication, but for systems which need to offer seamless integration with the normal world this approach may have to be leveraged with other development strategies. Both of these systems support trusted UI by implementing secure display and input drivers and display controllers to manage the secure framebuffers.

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 [25] 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 previously described systems. Because we are only interested in secure display, the following explanation discards the network delegation mode and input mechanism. 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 have corresponding 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 that memory region as secure only, thus isolating the framebuffer from rich OS manipulation.

Because this mechanism can still be victim of framebuffer overlay attacks, where a malicious backend driver gives a false framebuffer to the secure world, the system randomizes the background and foreground colours used in the display and uses two LEDs, controlled by the secure world, to show these same colours. A user can visually check if the colours shown in the secure LEDs match those of the display. If they match then the user is assured that the display shown is being controlled by the secure world.

These systems support secure display, as such, they do not disclose sensitive data. However, TrustOTP and TrustDump do not offer a fully integrated

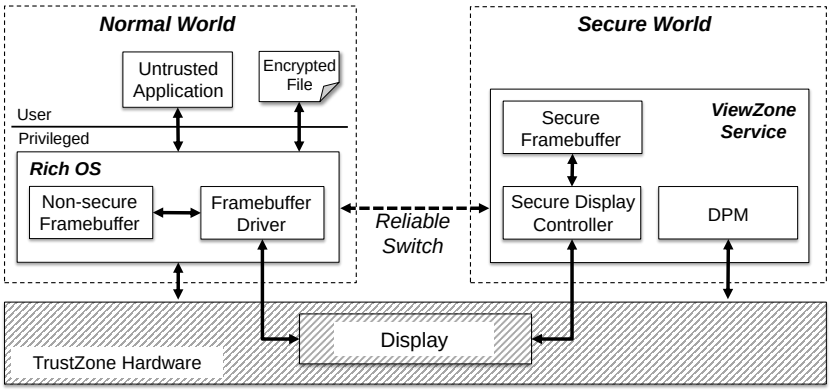

Figure 5. ViewZone architecture.

environment with the Android running in the normal domain. TrustUI, on the other hand, fully integrates its environment with that of the Android operating system. Furthermore, TrustUI describes a novel mechanism for the reuse of untrusted driver in the normal world without compromising security, which significantly reduces the secure system's TCB. The main disadvantage of TrustUI is that, similarly to TrustOTP and TrustDump, does not support the development of generic display applications. Besides, TrustUI is not immune to denial-of-service attacks by a malicious operating system running in the normal world, which may compromise the execution of the secure system.

From the systems described in this section we learned the underlying strategies for developing security systems using TrustZone. For this reason, the following section describes the architecture of *ViewZone*, a solution for securing the output of Android applications using TrustZone.

3 Architecture

This section presents *ViewZone*, our proposed solution for TrustZone-enabled devices which supports trusted display for critical Android applications which handle privacy sensitive data. Protected application data may take several formats, either being images or straight plain text files. This flexibility allows app developers to easily create several basic secure applications such as a secure password viewer, a sensitive image display and a sensitive document viewer, without ever being at risk of disclosure by a compromised full-featured OS.

Figure 5 represents the components and mechanisms which comprise *ViewZone*'s architecture. A typical execution flow of a *ViewZone* applications is as follows: an untrusted application running in the rich OS uses *ViewZone* as a trusted service in the secure domain. This untrusted app loads an encrypted file, whose content is only accessible by the secure world, into a shared memory

region. This allows the Data Processing Module (DPM) to securely copy the encrypted file to secure memory so it can be decrypted and later displayed to the user. The DPM looks into the shared memory region only when the reliable switch is triggered by a secure Non-maskable Interrupts (NMI) hardware interrupt, thus assuring the subsequent operations are executed by the secure side and the content displayed is authentic.

Once the DPM has access to the sensitive file content, the secure display controller copies the non-secure framebuffer to the secure framebuffer and the content of the sensitive file is integrated with that of the original framebuffer, giving the impression of integrated user interface between both domains. This means the trusted service can display content as it would be displayed by the original rich OS interface. When another NMI interrupt is triggered, the secure framebuffer is cleared and the control returns to the rich OS.

In order to support secure display as a service for Android applications and maintain a small trusted computing base, *ViewZone* must feature only the strictly necessary components for it to work, and even these components must be designed and built with intelligent and conservative development strategies. Moreover, *ViewZone* must allow for an integrated environment with Android to provide a transparent security environment for critical 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 trusted display, confidentiality, integrity and authenticity of data stored in the untrusted world, and be developer friendly in order to be easily deployable. The following paragraphs describe how these goals will be supported in *ViewZone*.

Trusted User Interface Trusted services must support secure display so the data from 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, *ViewZone* can prevent potential data leakage, as the secure framebuffer is reserved for use in the secure world. Moreover, because resources such as the video card and display screen are usually shared by both domains, the reliable display controller must be able to correctly program both resources. With this mechanism there is a guarantee that the information displayed to the user is authentic.

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 OS 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 [44] and TrustDump [43] in these systems. NMI are hardware interrupts that cannot be ignored by standard interrupt masking techniques. These are typically used to signal attention for non-recoverable hardware errors, but 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 *ViewZone* explicitly triggers the switch.

Secure File Handling In order for the sensitive information to be inaccessible by the rich operating system, the file which contains such information must be encrypted. This encryption must be leveraged so that the file is not understandable by the full-featured operating system but is readable by the *ViewZone* service. To achieve this, *ViewZone* must support a unique private key from factory which, paired with a known public key, allows for files to be encrypted before being stored in the untrusted domain. *ViewZone* must also support cryptographic protocols for key renewal. Key management and file handling, which includes copying a sensitive file from a shared memory region into secure memory and decrypting such file, is done by a *ViewZone* module designated Data Processing Module (DPM).

Developer Friendly *ViewZone* should allow for an easy deployment of critical applications using the trusted display service. The developer should not have to handle low level primitives regarding device drives, framebuffers or display controllers, as this is to be done by *ViewZone*. As such, developers should leverage an API which, by communicating with *ViewZone*'s service running in the secure domain, is capable of registering for the secure service and send the encrypted file to be displayed by the secure service.

4 Implementation and Evaluation

This section describes the implementation details of the solution as well as how the system is going to be evaluated. *ViewZone* will run along side Android and will be implemented upon a Freescale i.MX53 Quick Start Board development board [17], which supports ARM's TrustZone. To fully demonstrate the concept of *ViewZone*, we aim to develop a sensitive mobile health application for displaying Personal Health Records (PHR). This application allows a user to display and store his personal health information securely, without sensitive information ever being exposed to an untrusted rich OS such as Android.

4.1 Implementation

The PHR application comprises an untrusted client Android application which loads the sensitive encrypted file to a dedicated memory region shared by both worlds. This file is the personal health record to be displayed to the user. The client application then notifies the user the file is loaded and the user may proceed with the reliable world switch by pressing the button dedicated for the NMI interrupt. After this world switch, the file in the shared memory region is

decrypted and shown to the user. The file can then be stored securely in the untrusted world by saving it and triggering a new reliable world switch, which encrypts the file and stores it in the untrusted rich OS.

Secure Display is supported by a self-contained secure display controller in the secure domain and an Image Processing Unit (IPU) driver implemented in the rich OS. When the rich OS is running, the IPU is set as a non-secure device and can transfer data from the non-secure framebuffer to the display. When the system switches to the secure domain, the secure display controller checks the integrity of the IPU driver, saves its state and resets it as a secure device in order to transfer the data from the secure framebuffer to the display device. Before switching back to the normal world, the controller erases the footprint in the IPU to prevent information leakage, and then restores the device state for the rich OS. This method, which reuses the IPU driver implemented by the rich OS, needs additional code to check the driver's integrity, but maintains a smaller TCB than if a self-contained IPU driver was implemented in the secure world.

Memory Isolation is possible through a watermarking mechanism available in the i.MX53 QSB. This mechanisms allows the isolation of secure memory regions from non-secure memory ones designated for the rich OS. This memory isolation is necessary for the implementation of the secure framebuffer and to isolate the decrypted file from the normal world domain after the reliable world switch. The i.MX53 QSB has two banks of RAM, each with 512 MB and the watermarking mechanism can watermark one continuous region of up to 256 MB on each bank, totalling 512 MB of possible secure RAM. This is more than enough to support the framebuffer and most regular sized files.

Reliable Switch is based on the non-maskable interrupt mechanism, but the i.MX53 QSB does not provide any NMI explicitly. For this reason an NMI needs to be constructed in order to support reliable world switches. To achieve this the interrupt type of the NMI must be assigned as secure in the Interrupt Security Register (TZIC_INTSEC), which prevents the rich OS to modify the configuration of the NMI. Then, several configuration bits are set to zero so that the rich OS cannot disable, block or intercept the interrupt request made to the ARM processor. Finally the interrupt source, such as a physical button, must be configured as a secure peripheral.

4.2 Evaluation

To evaluate *ViewZone* we will measure the performance overhead for displaying files, as well as compare the development process for the PHR mHealth app using ViewZone 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 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 *ViewZone* with the value measured in the same conditions for a similar application without using the secure system. We will also assess the complexity of *ViewZone*'s TCB and the final attack surface.

5 Future Work Plan

Future work is scheduled as follows:

- January 9 - March 25: Fully design and implement the proposed architecture;
- March 26 - May 1: Perform a complete experimental evaluation;
- March 26 - May 10: Write a paper describing the project;
- May 11 - June 15: Write the dissertation;
- June 15: Deliver the MSc dissertation.

6 Conclusions

This project aims at developing a new security system for mobile applications named *ViewZone*. With this solution, apps can display sensitive data to the user without exposing such information to the OS. This is achieved by using ARM's TrustZone technology, which allows for the execution of isolated environments in TrustZone-enabled mobile devices, thus mitigating information leakage problems inherent to shared resources. Using *ViewZone*, Android applications can use *ViewZone*'s trusted service running in the isolated secure domain in order to securely display sensitive information. These trusted service can securely control the display peripherals and drivers by implementing a self-contained secure display controller and a secure isolated framebuffer. Moreover, unlike previous systems, *ViewZone* mitigates denial-of-service attacks by relying on secure hardware interrupts for a reliable world-switch. In a near future, and in order to evaluate this new security framework, an instance of *ViewZone* and of a secure personal health record mobile application will be implemented. This implementation will allow to assess both *ViewZone*'s performance in real life scenarios as well as its usefulness in the development of critical applications.

References

1. Android DRM Framework. http://developer.android.com/reference/android/drm/package-summary.html. Accessed: 2016-01-07.
2. Android Key Store. http://developer.android.com/training/articles/keystore.html. Accessed: 2016-01-07.
3. Sasikanth Avancha, Amit Baxi, and David Kotz. Privacy in mobile technology for personal healthcare. *ACM Computing Surveys (CSUR)*, 45(1):3, 2012.
4. 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.
5. 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.
6. Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating smart personal devices in restricted spaces.
7. 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.
8. 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.
9. 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.
10. Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Information Security*, pages 331–345. Springer, 2011.
11. Miguel Costa. SecMos - Mobile Operating System Security. Master's thesis, Instituto Superior Técnico, Portugal, 2015.
12. 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.
13. OMA DRM. Open mobile alliance digital rights management.(2010). *Retrieved May*, 2, 2011.
14. Nuno Duarte. On The Effectiveness of Trust Leases in Securing Mobile Applications. Master's thesis, Instituto Superior Técnico, Portugal, 2015.
15. 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.
16. Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. The andix research os-arm trustzone meets industrial control systems security.
17. Freescale i.MX53 Quick Start Board development board. http://www.freescale.com/products/arm-processors/i.mx-applications-processors-based-on-arm-cores/i.mx53-processors/i.mx53-quick-start-board:IMX53QSB. Accessed: 2016-01-07.

18. Genode OS. http://genode.org/. Accessed: 2016-01-07.
19. GlobalPlatform. https://www.globalplatform.org/. Accessed: 2016-01-07.
20. 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.
21. 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.
22. 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.
23. Kari Kostiainen et al. On-board credentials: an open credential platform for mobile devices. 2012.
24. David Kotz. A threat taxonomy for mhealth privacy. In *COMSNETS*, pages 1–6, 2011.
25. 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.
26. 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.
27. 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.
28. 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.
29. 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/.
30. Brian McGillion, Tanel Dettenborn, Thomas Nyman, and N. Asokan. Open-TEE – an open virtual trusted execution environment. Technical report, Aalto University, 2015.
31. 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.
32. 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.
33. OP-TEE. https://wiki.linaro.org/WorkingGroups/Security/OP-TEE. Accessed: 2016-01-07.
34. 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.

35. PolarSSL. https://polarssl.org/. Accessed: 2016-01-07.
36. Research2Guidance. Mobile Health Market Report 2013-2017. Technical report, Research2Guidance, 03 2013. http://research2guidance.com/.
37. Reuters - "Your medical record is worth more to hackers than your credit card". http://www.reuters.com/article/2014/09/24/us-cybersecurity-hospitals-idUSKCN0HJ21I20140924. Accessed: 2016-01-07.
38. 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.
39. 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.
40. Bilal Shebaro, Oluwatosin Ogunwuyi, Daniele Midi, and Elisa Bertino. Identidroid: Android can finally wear its anonymous suit. 2014.
41. SierraTEE. http://www.openvirtualization.org/. Accessed: 2016-01-07.
42. Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.
43. 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.
44. 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.
45. 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.
46. T6 TEE. https://www.trustkernel.com/products/tee/t6.html. Accessed: 2016-01-07.
47. TLK. http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/papers/webcrypto2014_submission_25.pdf. Accessed: 2016-01-07.
48. 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.
49. 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.
50. 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.

Table of Contents