ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Π.Μ.Σ. «Τεχνοοικονομική Διοίκηση και Ασφάλεια Ψηφιακών Συστημάτων»

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΘΕΜΑ:

«APPLICATION DEVELOPMENT IN THE TRUSTED EXECUTION ENVIRONMENT»

Σπουδαστής:

ΑΓΓΕΛΑΚΗΣ ΕΜΜΑΝΟΥΗΛ

Επιβλέπων Καθηγητής:

Δρ. ΞΕΝΑΚΗΣ ΧΡΗΣΤΟΣ

ΦΕΒΡΟΥΑΡΙΟΣ 2018

ABSTRACT

Almost a decade ago, the first phones appeared with hardware based TEEs, reaching today, all modern mobile devices contain a TEE.
Despite the large-scale deployment, the use of TEEs functionalities has been limited to mobile device constructors and a closed community to develop applications for them. Moreover, the use of hardware-based TEEs in application development and research are expensive and often proprietary.
Nowadays, many industry associations like GlobalPlatform are working to standardize the specifications of the TEEs, so it is possible for any developer to create an application for TEEs, with a predefined standardization in a virtual trusted environment. This will help developers and researchers, to enhance the protection and functionality to new applications and services.
This thesis deals with the development of a Trusted Application in a virtual Trusted Execution Environment, that makes use of all the security features this architecture provides. All of the tools and the development environment that are used, are common and well-known to the -developer community. Finally, we would like to underline that open virtual TEE provides the ability to developers and researchers, of applications and services that have been developed, to be tested, refined and the continuation of development.

ΕΥΧΑΡΙΣΤΙΕΣ

Με την περάτωση της παρούσας Μεταπτυχικής Διπλωματικής Εργασίας θα ήθελα να ευχαριστήσω θερμά τον Καθηγητή του Τμήματος Ψηφιακών Συστημάτων του Πανεπιστημίου Πειραιώς κ. Χρήστο Ξενάκη για την καθοριστική συμβολή του στην εκπόνηση της παρούσας Μεταπτυχικής Διπλωματικής Εργασίας.

Ακόμα θα ήθελα να ευχαριστήσω όλους τους καθηγητές του προγράμματος «Ασφάλεια Ψηφιακών Συστημάτων», για τις γνώσεις που μου μετέδωσαν οι οποίες υπήρξαν καθοριστηκές στο πλαίσιο εκπόνησης της παρούσας εργασίας.

Ιδιαίτερες ευχαριστίες θα ήθελα να δώσω στο συνάδελφο Λεωνίδα Περρωτή, για την άψογη συνεργασία που αναπτύξαμε στα πλαίσια της ανάπτυξης της εφαρμογής.

Τέλος θα ήθελα να ευχαριστήσω θερμά την οικογένεια μου για την καθοδήγηση, την συμπαράσταση και για όλα όσα μου έχουν προσφέρει μέχρι σήμερα.

Contents

LIST OF ABBREVIATIONS

API - APPLICATION PROGRAMMING INTERFACE

BYOD - BRING YOUR OWN DEVICE

CA - CLIENT APPLICATION

DRM - DIGITAL RIGHTS MANAGEMENT

DRM - DIGITAL RIGHTS MANAGEMENTS

EAL2+ - EVALUATION ASSURANCE LEVEL

FIDO - FAST IDENTITY ONLINE

GP- GLOBAL PLATFORM

HD - HIGH DEFINITION

HMAC - KEYED-HASH MESSAGE AUTHENTICATION CODE

HSM - HARDWARE SECURITY MODULES

IPC - INTERPROCESS COMMUNICATION

MNO - MOBILE NETWORK OPERATOR

NFC - NEAR FIELD COMMUNICATION

NVM - NON-VOLATILE MEMORY

OEM - ORIGINAL EQUIPMENT MANUFACTURERS

OMTP - OPEN MOBILE TERMINAL PLATFORM

OS - OPERATING SYSTEM

PIN - PERSONAL IDENTIFICATION NUMBER

POS - POINT OF SALE

RAM - RANDOM ACCESS MEMORY

REE - RICH EXECUTION ENVIRONMENT

ROM - READ ONLY MEMORY

RPMB - REPLAY PROTECTED MEDIA BLOCK

SE - SECURE ELEMENT

TA - TRUSTED APPLICATION

TEE - TRUSTED EXECUTION ENVIRONMENT

TPM - TRUSTED PLATFORM MODULE

UI - USER INTERFACE

UUID	UNIVERSALLY UNIQUE IDENTIFIER
WAC	WHOLESALE APPLICATIONS COMMUNITY
eMMC	EMBEDDED MULTIMEDIA CARD

1. INTRODUCTION

Personal computing devices such as smartphones, tablets and laptops have become pervasive. They are used to store sensitive data and access critical services across a wide range of domains, such as banking, health care and safety, where privacy and security are paramount. On the other hand, traditional operating systems and the services that they provide are becoming so large and complex that the task of securing them is increasingly harder. Hardware-based trusted execution environments (TEEs) were developed to address this gap. A TEE on a device is isolated from its main operating environment by using hardware security features. It offers a smaller operating environment that provides just enough functionality so that sensitive data and operations can be offloaded to it.

Hardware-based TEEs have been widely deployed in mobile devices for over a decade [1]. TI M-Shield [2] and ARM TrustZone [3], [4] are early examples, followed by newer architectures like the Intel SEP security co-processor [5] and Apple's "Secure Enclave" co-processor [6]. Business requirements such as the need to enforce digital rights management and subsidy locks, as well as regulatory requirements like cloning- and theft protection have been the driving forces behind such large scale deployment [1]. Such requirements continue to appear: e.g., fingerprint scanners with hardware protection, hardware-backed keystores, and the recent "kill switch" [7] bill in California mandating that a mobile device must be capable of being rendered inoperable if it is stolen.

Although the early hardware security modules (HSMs) like the IBM cryptocards2 were programmable [8], the vast majority of HSMs used with personal computers and servers today are typically application-specific modules or fixed function co- processors like the Trusted Platform Modules (TPMs) [9]. In contrast, TEEs in mobile devices are programmable. However, despite widespread deployment of hardware-based TEEs in mobile devices, application developers have lacked the interfaces to use TEE functionality to protect their applications and services. Nor have they been researched extensively in the academic community. Recent efforts by GlobalPlatform [10] to specify standard interfaces for TEE functionality in mobile devices [11] will partially address this problem. However, there are a number of factors that stand in the way of widespread use of hardware-based TEEs in application development and research. Chief among them is the difficulty of developing applications for TEEs. Software development kits for TEE application development are often proprietary or expensive. Debugging low-level TEE applications either requires expensive hardware debugging tools, or leaves the developer with only primitive debugging techniques like "print tracing" (e.g., using printf statements in C to keep track of how values of variables change during program execution).

1.1 TEE

The TEE is a secure area of the main processor in a smart phone (or any connected device). It ensures that sensitive data is stored, processed and protected in an isolated, trusted environment. The TEE's ability to offer isolated safe execution of authorized security software, known as 'trusted applications', enables it to provide end-to-end security by enforcing protected execution of authenticated code, confidentiality, authenticity, privacy, system integrity and data access rights. Comparative to other security environments on the device, the TEE also offers high processing speeds and a large amount of accessible memory.
The TEE offers a level of protection against attacks that have been generated in the Rich OS environment. It assists in the control of access rights and houses sensitive applications, which need to be isolated from the Rich OS. For example, the TEE is the ideal environment for content providers offering a video for a limited period of time, as premium content (e.g. HD video) must be secured so that it cannot be shared for free.

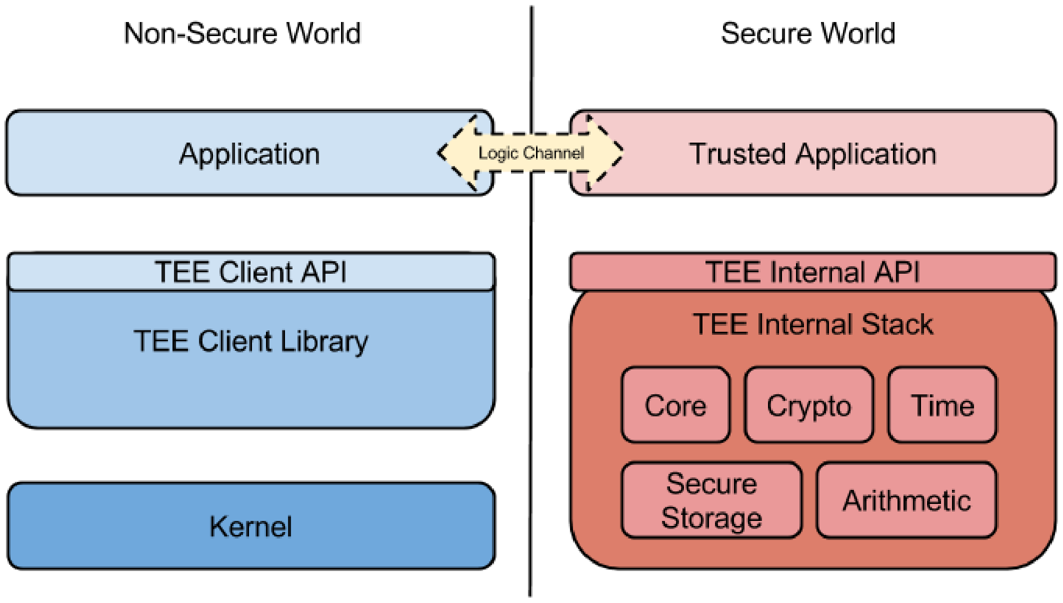


Figure 1.1 -1 TEE Archiitecture

1.1.1 Rich Execution Environment (REE)

“Rich” refers to an operating environment that is feature rich as would expect from modern platforms such as Windows, Linux, Android, ios, etc. In this environment the majority of applications are being developed and deployed.

1.1.2 Trusted Execution Environment (TEE)

TEE is a combination of features, both software and hardware, that isolate it work done by REE. These environments have a limited set of features and services, since they are only intended to address the critical security subsystem such as unloading some cryptographic features or keys Management.

1.1.3 Trusted Application (TA)

An application encapsulating the security-critical functionality to be run within the TEE. This may be a service style application that provides a general feature, such as a generic cryptographic KeyStore, or it could be designed to offload a very specific part of an application that is running in the REE, such as a portion of the client state machine in a security protocol like TLS.

1.1.4 Client Application (CA)

CAs are ordinary applications (e.g. browser or e-mail client) running in the REE. CAs are responsible for providing the majority of an application’s functionality but can invoke TAs to offload sensitive operations. Examining a typical TEE application workflow sequence, using GP terminology, let’s consider a common use case for TEEs: the offloading of DRM protected content. The CA would be responsible for the majority of the tasks associated with viewing the content i.e. opening the media file, providing a region in the display into which it can be rendered (the window) and providing a mechanism by which to start, stop and rewind the media. A TA would be used to decrypt the protected media stream and make the decrypted content available directly to the graphics hardware that is responsible for rendering and displaying the stream.

1.2 Benefits of using a TEE

From a business and commercial perspective, the TEE meets the requirements of all of the key players. At a high level:

- **Mobile manufacturers'** security concerns are tied to several factors, not the least of which being the sheer number of stakeholders involved in device and application delivery. A framework (such as GlobalPlatform-certified TEE) that guarantees a minimum baseline for platform security would allow all stakeholders to make updates to devices and applications while minimizing threats to consumers.
- **For MNOs** the TEE delivers a higher level of security than what the Rich OS offers and higher performance than what a Secure Element (SE) typically offers. In essence, the TEE ensures a high level of trust between the device, the network, the edge and the cloud, thereby improving the ability of a MNO to enhance services for root detection, SIM-lock, anti-tethering, mobile wallet, mobile as PoS, data protection, mobile device management, application security, content protection, device wipes, and anti-malware protection.
- **Content and service providers** want the TEE to ensure that their product remains secure and can be deployed to numerous platforms in a common manner and is easily accessible to the end user.
- **Payment service providers** do not want to have to develop different versions of the same application to satisfy the needs of different proprietary TEE environments. E.g. if the ecosystem is not standardized, payment service providers will have to be certified and support different applications and processes. This is time consuming, costly and counterintuitive to the goal of creating a mass market for application deployment.

Focusing specifically on security, the TEE is a unique environment that is capable of increasing the security and assurance level of services and applications, in the following ways:

- **User Authentication:** Using the trusted UI, the TEE makes it possible to securely collect a user's password or PIN. This trusted user authentication can be used to verify a cardholder for payment, confirm a user's identification to a corporate server, attest to a user's rights with a content server, and more.

- **Trusted Processing and Isolation:** Application processing can be isolated from software attacks by running in the TEE. Examples include processing a payment, decrypting premium content, reviewing corporate data, and more.
- **Transaction Validation:** Using the trusted UI, the TEE ensures that the information displayed on-screen is accurate. This is useful for a variety of functions, including payment validation or protection of a corporate document.
- **Usage of Secure Resources:** By using the TEE APIs, application developers can easily make use of the complex security functions made available by a device's hardware, instead of using less safe software functions. This includes hardware cryptography accelerators, SEs, biometric equipment and the secure clock.
- **Certification:** Trusted certification is best achieved through standardization of the TEE, which in turn improves stakeholder confidence that the security-dependent applications are running on a trusted platform.

1.3 TEE into the security infrastructure of a smartphone

It is useful to put the TEE in the context of the overall security infrastructure of a mobile device. There are three environments which make up the framework. Each has a different task:

- **Rich OS:** An environment created for versatility and richness where device applications, such as Android, Symbian OS, and Windows Phone for example, are executed. It is open to third party download after the device is manufactured. Security is a concern here but is secondary to other issues.
- **TEE:** The TEE is a secure area of the main processor in a smartphone (or any connected device) and ensures that sensitive data is stored, processed and protected in an isolated, trusted environment. The TEE's ability to offer isolated safe execution of authorized security software, known as 'trusted applications', enables it to provide end-to-end security by enforcing protection, confidentiality, integrity and data access rights. The TEE offers a level of protection against software attacks, generated in the Rich OS environment. It assists in the control of access rights and houses sensitive applications, which need to be isolated from the Rich OS. For example, the TEE is the ideal

environment for content providers offering a video for a limited period of time that need to keep their premium content (e.g. HD video) secure so that it cannot be shared for free.

- **SE:** The SE is a secure component which comprises autonomous, tamper-resistant hardware within which secure applications and their confidential cryptographic data (e.g. key management) are stored and executed. It allows high levels of security, but limited functionality, and can work in tandem with the TEE. The SE is used for hosting proximity payment applications or official electronic signatures where the highest level of security is required. The TEE can be used to filter access to applications stored directly on the SE to act as a buffer for Malware attacks.

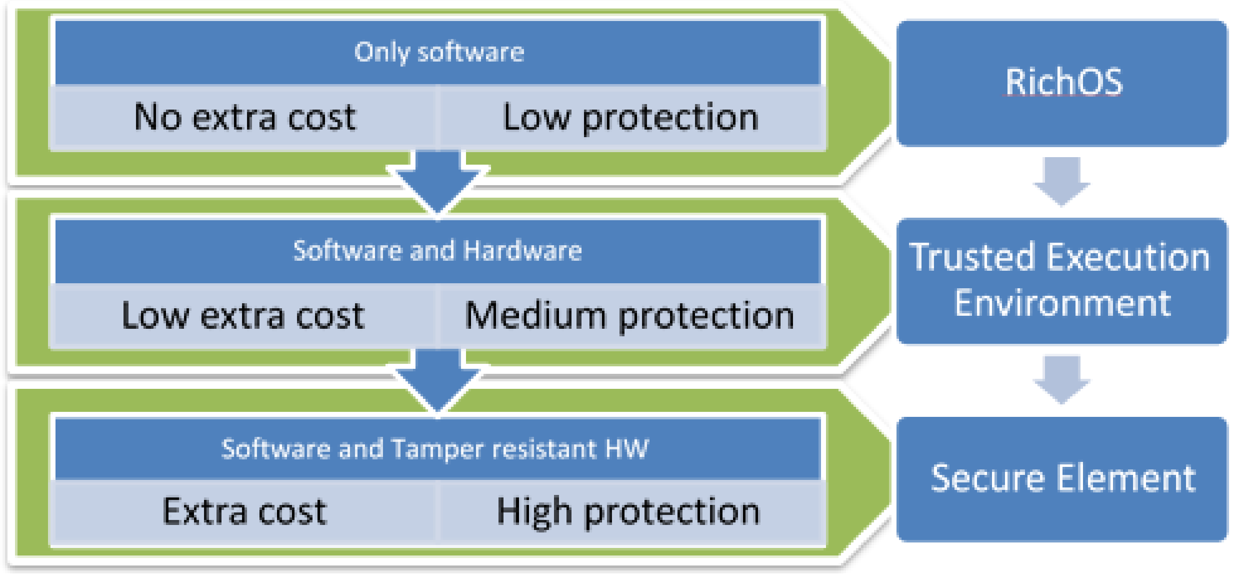


Figure 1.3-2 Smartphone Architecture cost-protection comparison

The Rich OS is therefore a rich environment that is vulnerable to both software and physical attacks. The SE, on the other hand, is resilient to physical attacks but somewhat constrained in execution processing capabilities. The TEE, however, serves as an ideal balance between Rich OS performance and SE security, and a companion to both. The security offered by the TEE, in general, is sufficient for most applications. Moreover, the TEE provides a more powerful processing speed capability and greater accessible memory space than an SE (these are, in fact, quite similar to that of a Rich OS).

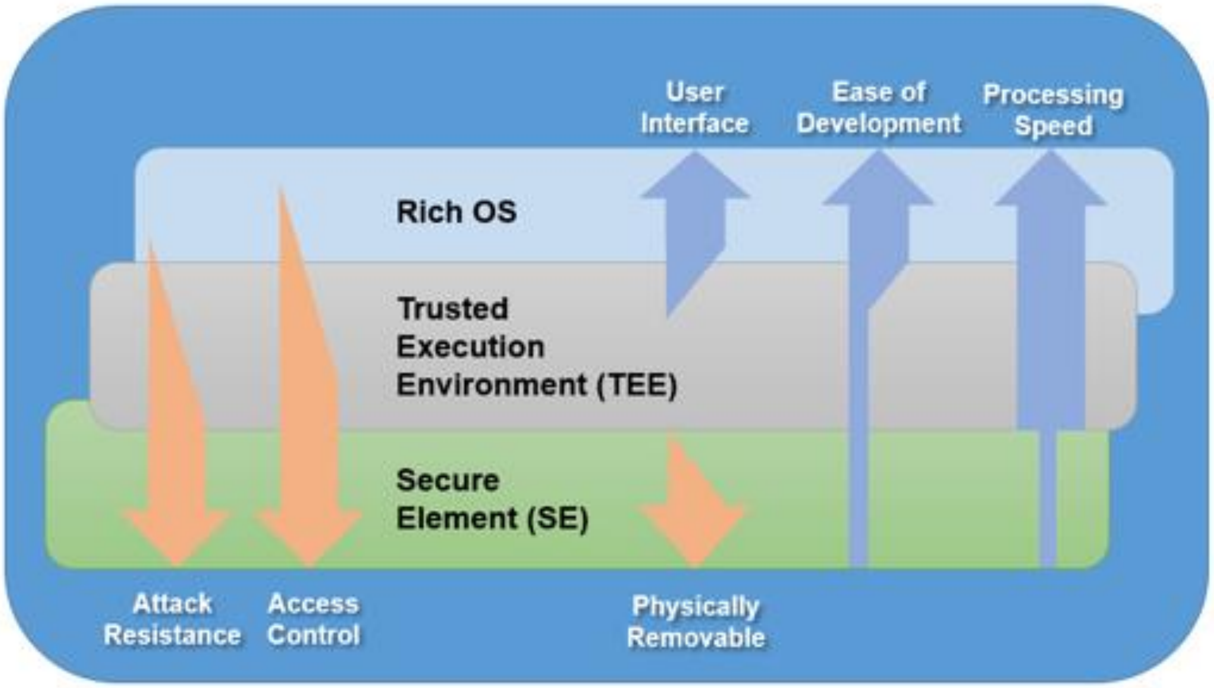


Figure 1.3-3 Smartphone Architecture and remarkable points on layers

2. STANDARDIZATION & GLOBAL PLATFORM

The two main reasons why the TEE was invented and developed are, firstly that the number of mobile devices is increasing globally and rapidly, and those devices require a greater level of security and -more importantly- the continuous digitalization of more aspects of every day's life that lead to more users being online and more applications being developed, thus the need of security measures such as TEE is increased.

The main functionality of TEE is that it completely isolates trusted applications from Rich OS and all the potential threats that may exist in Rich OS, such as downloaded malwares. It is safe to say that the TEE is going to become a main component of every device as the secure services market evolves.

However, the resulting lack of standardization of TEE has presented application developers with a significant challenge to overcome; each proprietary TEE solution requires a different version of the same application to ensure that the application conforms to unique versions of the technology. In addition, if the application provider wishes to deploy to multiple TEE solution environments and have assurance that each environment will provide a common level of security, then a security evaluation will need to be performed on each TEE solution. This leads to a resource intensive development process.

GlobalPlatform is an initiative whose purpose is the standardization of TEE. Providing one standard, regarding TEE environment has become a need because of the existence of many proprietary systems each describing a different TEE architecture. Standardizing the TEE addresses problems such us, the need of specialized skills and the cost of integration and implementation of one application to the many different platforms. Thus, GloabalPlatform specification becomes a powerful tool for implementers of mobile wallets, NFC payment implementations, premium content protection and bring your own device (BYOD) initiatives.

Since GlobalPlatform is handset and Rich OS agnostic, it is well placed to bring forward specifications for the TEE that can be embraced by all suppliers and reside comfortably alongside each of their rich OS environments. Interoperability in both functionality and security will be enhanced by the standardization of the TEE. This will simplify application development and deployment for all concerned, saving costs and time to market.

Trustonic was one of the first companies to qualify a GlobalPlatform-compliant TEE product. Since then, a significant number of GlobalPlatform TEE implementations have become available. A list of those which have been formally qualified by GlobalPlatform can be found at, and many other TEE products offer a high level of compatibility with GlobalPlatform standards.

The following list provides a reference and a short description to all of the GlobalPlatform APIs, quoted by the official GlobalPlatform website:

TEE Client API Specification v1.0	Communication between applications running in a mobile OS and trusted applications residing in the TEE.
TEE Systems Architecture v1.0	Hardware and software architectures behind the TEE.
TEE Internal API Specification v1.0	Development of trusted applications.
TEE Secure Element API Specification v1.0	Syntax and semantics of the TEE Secure Element API. It is suitable for software developers implementing trusted applications running inside the TEE which need to expose an externally visible interface to client applications.
Trusted User Interface API Specification v1.0	Specifications of the trusted UI, how it should facilitate information that will be securely configured by the end user and securely controlled by the TEE.
TEE TA Debug Specification v1.0	GlobalPlatform TEE debug interfaces and protocols.
TEE Management Framework v1.0	GlobalPlatform Remote Administration Framework, which enables trusted applications on a device to be remotely managed by trusted service providers.

The GlobalPlatform TEE Protection Profile specifies the typical threats, the hardware and software of the TEE needs to withstand. It also details the security objectives that are to be met in order to counter these threats and the security functional requirements that a TEE will have to comply with.

A security assurance level of EAL2+ has been selected; the focus is on vulnerabilities that are subject to widespread, software-based exploitation.

The Common Criteria portal has officially listed the GlobalPlatform TEE Protection Profile on its website, under the Trusted Computing category. This important milestone means that industries using TEE technology to deliver services such as premium content and mobile wallets, or enterprises and governments establishing secure mobility solutions, can now formally request that TEE products are certified against this security framework.

Finally, GlobalPlatform is committed to ensuring a standardized level of security for embedded applications on secure chip technology. It has developed an open and thoroughly evaluated trusted execution environment (TEE) ecosystem with accredited laboratories and evaluated products. This certification scheme created to certify a TEE product in 3 months has been launched officially in June 2015.

3. OPEN-TEE

Open-TEE is an open source project whose main goal is to implement a virtual TEE compliant with the latest GP TEE specifications. The Open-TEE framework tools removing the need for specialized hardware, the overheads that it incurs and the elimination of cost for purchasing hardware TEEs. Furthermore, provides to allow developers access a TEE, even a virtual, that so far is restricted from chip and device manufacturers.

The most common debugging Trusted applications methods are either expensive or to resort to primitive print tracing by inserting diagnostic output in the source code, that provides detailed instruction level of debugging, but the costs associated with these debuggers can be prohibitively expensive and complex. Another problem encountered by the TEE developers is that if a TA, running on actual device hardware, crashes then a hard reset of the device may be required. Thereby the time and effort of the debugging process is significantly increased.

Safely exposing TEE functionality to application developers provides them with a means to improve the security and privacy of their applications. Also exposing TEE technology to a wider audience does not guarantee that all security threats will disappear, however, a large community can provide plentiful ideas and a more in-depth research, than a small group of people. The financial and technical aspects of accessing a hardware TEE make it infeasible, for this reason the creation of a TEE framework such as Open-TEE, that is not bound to any hardware or any particular vendor, yet tries to conform to one standardization effort (for which GP has been chosen) is a great improvement.

3.1 Architecture

The following section describes the design and implementation of such a software framework which is called Open-TEE. It will begin with an overview of the structure of the Open-TEE environment. Figure 3.1 identifies the main components and their relationships. The color code used in Figure 3.1 is the same as that used for Figure 2.3 to make the correspondence between the Open-TEE implementation architecture and the GP conceptual architecture is clear. Each component is described in detail below.

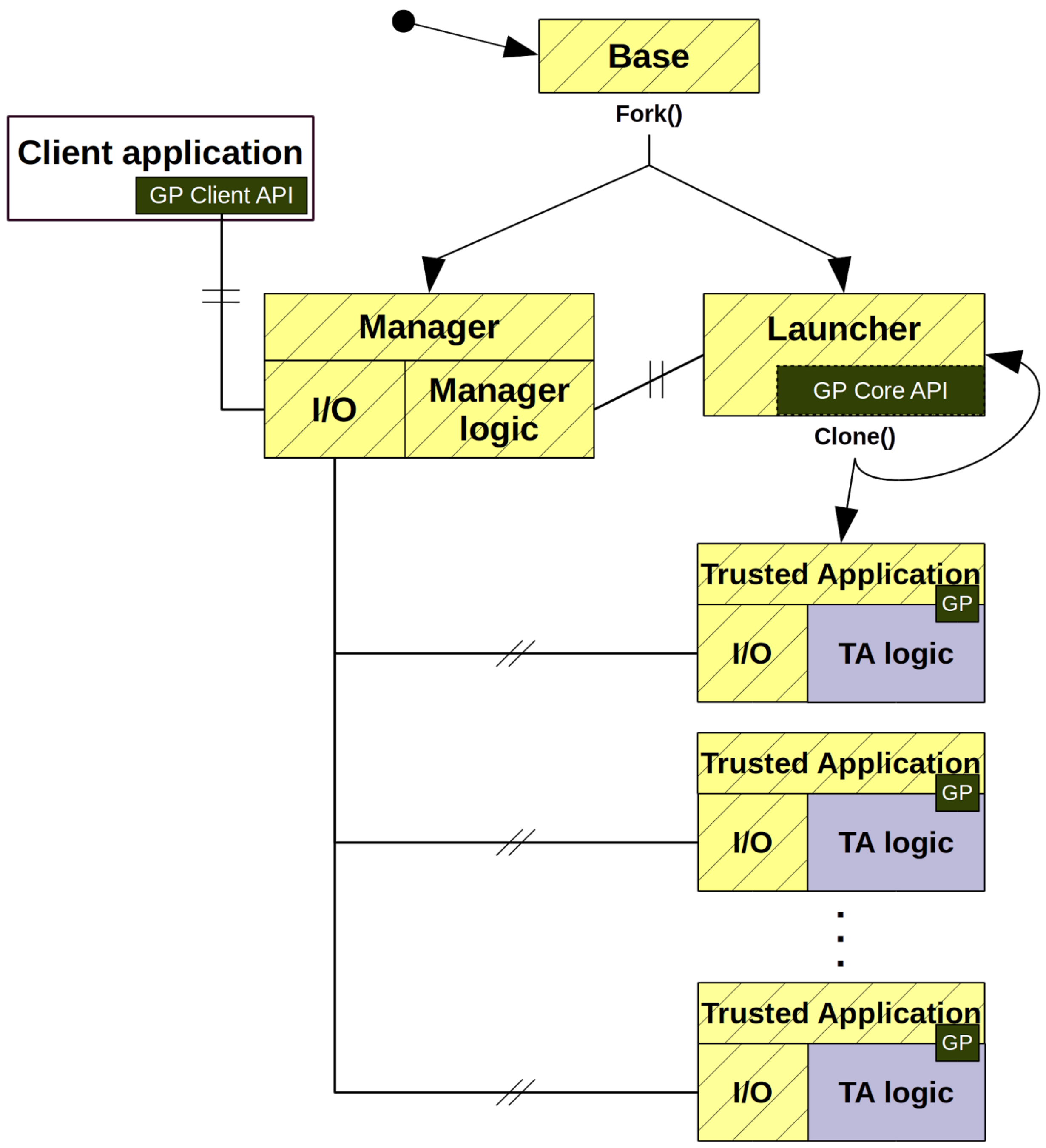


Figure 3.1-4 Open-TEE arcitecture

3.1.1 Base

Open-TEE is designed to function as a daemon process in user space. It starts executing Base, a process that encapsulates the TEE functionality. Base is responsible for loading the configuration and preparing the common parts of the system. Once initialized Base will fork and create two independent but related processes. One process becomes Manager and the other, Launcher which serves as a prototype for TAs.

3.1.2 Manager

Manager can be visualized as Open-TEE’s “operating system”. Its main responsibilities are: managing connections between applications, monitoring TA state, providing secure storage for a TA and controlling shared memory regions for the connected applications. Centralizing this functionality into a control process can also be seen as a wrapper abstracting the running environment (e.g. GNU/Linux) and reconciling it with the requirements imposed by the GP TEE standards. GP requirements and the host environment’s functionality are not always aligned. For example, GP requirements stipulate that if a TA/CA process crashes unexpectedly, all shared resources of the connected processes must be released. In a typical running environment, this requires additional steps beyond just terminating the process. For example, all shared memory must be unregistered – this needs to be a distinct action from normal process termination.

3.1.3 Launcher

The sole purpose of Launcher is to create new TA processes efficiently. When it is first created, Launcher will load a shared library implementing the GP TEE Core API and will wait for further commands from Manager. Manager will signal Launcher when there is a need to launch a new TA (for example, when there is a request from a CA). Upon receiving the signal, Launcher will clone itself. The clone will then load the shared library corresponding to the requested TA. The design of Launcher follows the “zygote” design pattern (such as that used in Android) of preloading common components. This is intended to improve the perceived performance of starting a new TA in Open-TEE: because shared libraries and configurations common to all TAs are preloaded into Launcher, the time required to start and configure the new process is minimal. A newly created TA process is then re-parented onto Manager so that it is possible for it to control the TA (so that, for example, it can enforce the type of GP requirements discussed in the paragraph above).

3.1.4 TA Processes

The architecture of the TA processes is inspired by the multi-process architecture utilized in the Chromium Project. Each process has been divided into two threads. The first handles Inter-Process Communication (IPC) and the second is the working thread, referred to respectively as the I/O and TA Logic threads. This architectural model enables the process to be interrupted without halting it, as occurs when changing status flags and adding new tasks to the task queue. Additional The architecture of Manager follows the same division benefits of this model are that it allows greater separation and abstraction of the TA functionality from the Open-TEE framework.

3.1.5 GP TEE APIs

The GP TEE Client API and GP TEE Core API are implemented as shared libraries in order to reduce code and memory consumption. In addition, loading the GP TEE Core API into Launcher when it is created will help to reduce the startup times of the TA process removing the need to load it for every TA; this is one of the key benefits of the “zygote” design.

3.1.6 IPC

Open-TEE implements a communication protocol on top of Unix domain sockets and inter-process signals as the means to both control the system and transfer the messages between the CA and TA.

4. GLOBAL PLATFORM: INTERNAL APIs SPECIFICATION

On the following section the core functions, data structures and some key aspects of the Global Platform Internal APIs will be presented, focusing on the ones that were used for this application. It must be noted that all the following are part of the Global platform specification and as so it is of a more generic approach than the "OpenTee" TEE implementation. However there are minimal differences as the OpenTEE developer team and each similar platform have based their work on Global Platform standards.

There are two main API's for developing TEE applications. The first one is the Client API that describes all the functions and interfaces that a CA can use to interact with the TEE engine and its corresponding TA. The second one is the TEE Internal Core API Specification that describes all the standards regarding the TA. As mentioned earlier this thesis is focused on the TA component of the application and for this purpose only the Internal API's are going to be described extensively.

All the following specifications are quoted from the Global Platform Specification documents.

4.1 TEE Internal Core API Specification

4.1.1 TA Interface

Each Trusted Application exposes an interface (the TA interface) composed of a set of entry point functions that the Trusted Core Framework implementation calls to inform the TA about life-cycle changes and to relay communication between Clients and the TA. Once the Trusted Core Framework has called one of the TA entry points, the TA can make use of the TEE Internal Core API to access the facilities of the Trusted OS. Each Trusted Application is identified by a Universally Unique Identifier (UUID) as specified in [RFC 4122]. Each Trusted Application also comes with a set of Trusted Application Configuration Properties. These properties are used to configure the Trusted OS facilities exposed to the Trusted Application. Properties can also be used by the Trusted Application itself as a means of configuration.

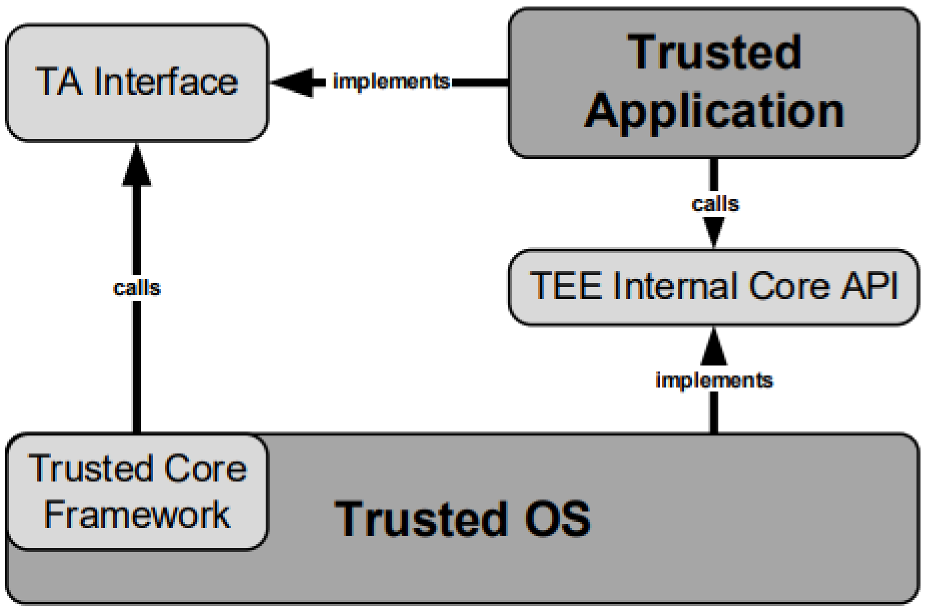


Figure 4.1-5 TEE Internal Architecture

TA_CreateEntryPoint	This is the Trusted Application constructor. It is called once and only once in the life-time of the Trusted Application instance. If this function fails, the instance is not created.	TEE_Result TA_EXPORT TA_CreateEntryPoint(void);
TA_DestroyEntryPoint	This is the Trusted Application destructor. The Trusted Core Framework calls this function just before the	void TA_EXPORT TA_DestroyEntryPoint(void);

	Trusted Application instance is terminated. The Framework MUST guarantee that no sessions are open when this function is called. When TA_DestroyEntryPoint returns, the Framework MUST collect all resources claimed by the Trusted Application instance	
TA_OpenSessionEntryPoint	This function is called whenever a client attempts to connect to the Trusted Application instance to open a new session. If this function returns an error, the connection is rejected and no new session is opened. In this function, the Trusted Application can attach an opaque void* context to the session. This context is recalled in all subsequent TA calls within the session.	TEE_Result TA_EXPORT TA_OpenSessionEntryPoint(uint32_t paramTypes, [inout] TEE_Param params[4], [out][ctx] void** sessionContext);

TA_CloseSessionEntryPoint	This function is called when the client closes a session and disconnects from the Trusted Application instance. The Implementation guarantees that there are no active commands in the session being closed. The session context reference is given back to the Trusted Application by the Framework. It is the responsibility of the Trusted Application to deallocate the session context if memory has been allocated for it.	void TA_EXPORT TA_CloseSessionEntryPoint([ctx] void* sessionContext);
TA_InvokeCommandEntryPoint	This function is called whenever a client invokes a Trusted Application command. The Framework gives back the session context reference to the Trusted Application in this function call.	TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint([ctx] void* sessionContext, uint32_t commandID, uint32_t paramTypes, [inout] TEE_Param params[4]);

Trusted Storage API for Data and Keys

Each TA has access to a set of Trusted Storage Spaces, identified by 32-bit Storage Identifiers. o The current version of this specification defines a single Trusted Storage Space for each TA, which is its own private storage space. The objects in this storage space are accessible only to the TA that created them and are not visible to other TAs.

A Trusted Storage Space contains Persistent Objects.

- Each persistent object is identified by an Object Identifier, which is a variable-length binary buffer from 0 to 64 bytes. Object identifiers can contain any bytes, including bytes corresponding to non-printable characters.
- A persistent object can be a Cryptographic Key Object, a Cryptographic Key-Pair Object, or a Data Object. Each persistent object has a type, which precisely defines the content of the object. For example, there are object types for AES keys, RSA key-pairs, data objects, etc.
- All persistent objects have an associated Data Stream. Persistent data objects have only a data stream. Persistent cryptographic objects (that is, keys or key-pairs) have a data stream, Object Attributes, and metadata.
- The Data Stream is entirely managed in the TA memory space. It can be loaded into a TA-allocated buffer when the object is opened or stored from a TA-allocated buffer when the object is created. It can also be accessed as a stream, so it can be used to store large amounts of data accessed by small chunks.
- Object Attributes are used for small amounts of data (typically a few tens or hundreds of bytes). They can be stored in a memory pool that is separated from the TA instance and some attributes may be hidden from the TA itself. Attributes are used to store the key material in a structured way. The metadata associated with each cryptographic object includes: Key Size in bits. The precise meaning depends on the key algorithm. For example, AES key can have 128 bits, 192 bits, or 256 bits; RSA keys can have 1024 bits or 2048 bits or any other supported size, etc. Key Usage Flags, which define the operations permitted with the key as well as whether the sensitive parts of the key material can be retrieved by the TA or not. A TA can also allocate Transient Objects.

Compared to persistent objects:

- Transient objects are held in memory and are automatically wiped and reclaimed when they are closed or when the TA instance is destroyed.
- Transient objects contain only attributes and no data stream.
- A transient object can be uninitialized, in which case it is an object container allocated with a certain object type and maximum size but with no attributes.
- A transient object becomes initialized when its attributes are populated. Note that persistent objects are always created initialized. This means that when the TA wants to generate or derive a persistent key, it has to first use a transient object then write the attributes of a transient object into a persistent object.
- Transient objects have no identifier, they are only manipulated through object handles. o Currently, transient objects are used for cryptographic keys and key-pairs. Any function that accesses a persistent object handle MAY return a status of TEE_ERROR_CORRUPT_OBJECT or TEE_ERROR_CORRUPT_OBJECT_2, which indicates that corruption of the object has been detected. Before this status is returned, the Implementation SHALL delete the corrupt object and SHALL close the associated handle; subsequent use of the handle SHALL cause a panic. Any function that accesses a persistent object MAY return a status of TEE_ERROR_STORAGE_NOT_AVAILABLE or TEE_ERROR_STORAGE_NOT_AVAILABLE_2, which indicates that the storage system in which the object is stored is not accessible for some reason. Persistent and transient objects are manipulated through opaque Object Handles.
- Some functions accept both types of object handles. For example, a cryptographic operation can be started with either a transient key handle or a persistent key handle. o Some functions accept only handles on transient objects. For example, populating the attributes of an object works only with a transient object because it requires an uninitialized object and persistent objects are always fully initialized. o Finally, the file-like API functions to access the data stream work only with persistent objects because transient objects have no data stream.

4.1.2 Cryptographic Operations API

This part of the Cryptographic API defines how to actually perform cryptographic operations: Cryptographic operations can be pre-allocated for a given operation type, algorithm, and key size. Resulting Cryptographic Operation Handles can be reused for multiple operations. When required by the operation, the Cryptographic Operation Key can be set up independently and reused for multiple operations. Note that some cryptographic algorithms, such as AES-XTS, require two keys. An operation may be in two states: initial state where nothing is going on and active state where an operation is in progress

The cryptographic algorithms:

Supported Algorithm Digests MD5 SHA-1 SHA-256 SHA-224 SHA-384 SHA-512

Symmetric ciphers DES Triple-DES with double-length and triple-length keys AES

Message Authentication Codes (MACs) DES-MAC AES-MAC AES-CMAC HMAC with one of the supported digests

Authenticated Encryption (AE) AES-CCM with support for Additional Authenticated Data (AAD) AES-GCM with support for Additional Authenticated Data (AAD)

Asymmetric Encryption Schemes RSA PKCS1-V1.5 RSA OAEP

Asymmetric Signature Schemes DSA RSA PKCS1-V1.5 RSA PSS

Key Exchange Algorithms Diffie-Hellman

DATA TYPES

TEE_UUID, TEEC_UUID	typedef struct { uint32_t timeLow; uint16_t timeMid; uint16_t timeHiAndVersion; uint8_t clockSeqAndNode[8]; } TEE_UUID;	TEE_UUID is the Universally Unique Resource Identifier type as defined in [RFC 4122]. This type is used to identify Trusted Applications and clients.

TEE_Result	typedef uint32_t TEE_Result;	
TEE_ObjectHandle	typedef struct __TEE_ObjectHandle* TEE_ObjectHandle;	TEE_ObjectHandle is an opaque handle on an object.a cryptographic object. TEE_ObjectHandle is an opaque handle on a cryptographic object. These handles are returned by the functions TEE_AllocateTransientObject.

4.1.3 Internal Core API: Most used functions

4.1.3.1 Generic Operation Functions

4.1.3.1.1 TEE_AllocateOperation

```
TEE_Result TEE_AllocateOperation(
TEE_OperationHandle* operation,
uint32_t algorithm,
uint32_t mode,
uint32_t maxKeySize );
```

4.1.3.1.2 TEE_ResetOperation

```
void TEE_ResetOperation( TEE_OperationHandle operation );
```

4.1.3.2 Memory Management Function

4.1.3.2.1 TEE_MemFill

```
void TEE_MemFill(
[outbuf(size)] void* buffer,
 uint32_t x,
uint32_t size);
```

4.1.3.2.2 TEE_MemMove

```
void TEE_MemMove(
[outbuf(size)] void* dest,
[inbuf(size)] void* src,
 uint32_t size );
```

4.1.3.2.3 TEE_MemCompare

```
int32_t TEE_MemCompare(
 [inbuf(size)] void* buffer1,
[inbuf(size)] void* buffer2,
 uint32_t size);
```

4.1.3.3 Message Digest Functions

4.1.3.3.1 TEE_DigestUpdate

```
void TEE_DigestUpdate(
TEE_OperationHandle operation,
[inbuf] void* chunk,
uint32_t size_t chunkSize )
```

4.1.3.3.2 TEE_DigestDoFinal

```
TEE_Result TEE_DigestDoFinal(
TEE_OperationHandle operation,
[inbuf] void* chunk,
uint32_t size_t chunkLen,
[outbuf] void* hash, uint32_t size_t *hashLen );
```

4.1.3.4 Persistent Object Functions

4.1.3.4.1 TEE_OpenPersistentObject

```
TEE_Result TEE_OpenPersistentObject(
uint32_t storageID,
[in(objectIDLength)] void* objectID,
uint32_t size_t objectIDLen,
uint32_t flags,
 [out] TEE_ObjectHandle* object );
```

Description The TEE_OpenPersistentObject function opens a handle on an existing persistent object. It returns a handle that can be used to access the object's attributes and data stream. The storageID parameter indicates which Trusted Storage Space to access. The flags parameter is a set of flags that controls the access rights and sharing permissions with which the object handle is opened. The value of the flags parameter is constructed by a bitwise-inclusive OR of flags from the following list:

4.1.3.4.2 TEE_CreatePersistentObject

```
TEE_Result TEE_CreatePersistentObject( uint32_t storageID,
[in(objectIDLength)] void* objectID,
 uint32_t size_t objectIDLen,
 uint32_t flags, TEE_ObjectHandle attributes,
 [inbuf] void* initialData, uint32_t size_t initialDataLen,
[out] TEE_ObjectHandle* object );
```

Description The TEE_CreatePersistentObject function creates a persistent object with initial attributes and an initial data stream content, and optionally returns either a handle on the created object, or TEE_HANDLE_NULL upon failure.

4.1.3.4.3 TEE_CloseAndDeletePersistentObject1

TEE_Result TEE_CloseAndDeletePersistentObject1(TEE_ObjectHandle object);

Description This function replaces the TEE_CloseAndDeletePersistentObject function, whose use is deprecated. The TEE_CloseAndDeletePersistentObject1 function marks an object for deletion and closes the object handle. The object handle MUST have been opened with the write-meta access right, which means access to the object is exclusive. Deleting an object is atomic; once this function returns, the object is definitely deleted and no more open handles for the object exist. This shall be the case even if the object or the storage containing it have become corrupted. The only reason this routine can fail is if the storage area containing the object becomes inaccessible (e.g. the user removes the media holding the object). In this case TEE_ERROR_STORAGE_NOT_AVAILABLE SHALL be returned. If object is TEE_HANDLE_NULL, the function does nothing.

4.1.3.5 Data Stream Access Functions

4.1.3.5.1 TEE_ReadObjectData

```
TEE_Result TEE_ReadObjectData(
TEE_ObjectHandle object,
 [out] void* buffer, uint32_t size_t size,
[out] uint32_t* count );
```

Description The TEE_ReadObjectData function attempts to read size bytes from the data stream associated with the object into the buffer pointed to by buffer. The object handle MUST have been opened with the read access right. The bytes are read starting at the position in the data stream currently stored in the object handle. The handle's position is incremented by the number of bytes actually read. On completion TEE_ReadObjectData sets the number of bytes actually read in the uint32_t pointed to by count. The value written to *count may be less than size if the number of bytes until the end-of stream is less than size. It is set to 0 if the position at the start of the read operation is at or beyond the end-of-stream. These are the only cases where *count may be less than size. No data transfer can occur past the current end of stream. If an attempt is made to read past the end-of-stream, the TEE_ReadObjectData function stops reading data at the end-of-stream and returns the data read up to that point. This is still a success. The position indicator is then set at the end-of-stream. If the position is at, or past, the end of the data when this function is called, then no bytes are copied to *buffer and *count is set to 0.

4.1.4 Effect of Client Operation on TA Interface

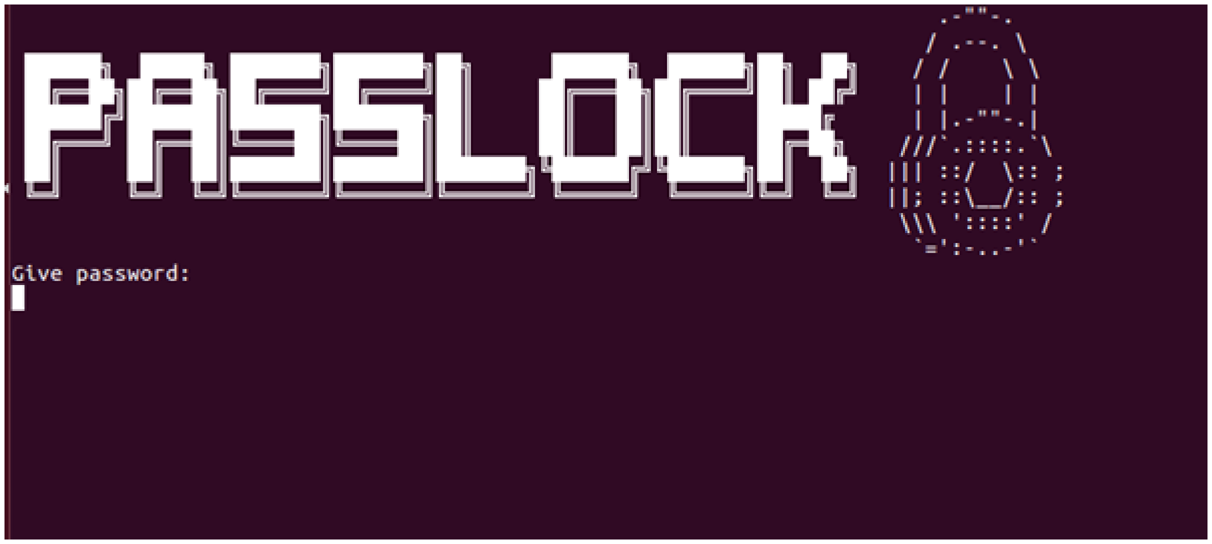
As for the Client application API the following table apposes all the basic functionalities, provided from the Client API Specification document, that trigger the execution of specific actions from the TA.

Client Operation	Trusted Application Effect
TEEC_OpenSession	If a new Trusted Application instance is needed to handle the session, TA_CreateEntryPoint is called. Then, TA_OpenSessionEntryPoint is called
TEEC_InvokeCommand	TA_InvokeCommandEntryPoint is called.
TEEC_CloseSession	TA_CloseSessionEntryPoint is called. For a multi-instance TA or for a single-instance, non keep-alive TA, if the session closed was the last session on the instance, then TA_DestroyEntryPoint is called. Otherwise, the instance is kept until the TEE shuts down
TEEC_RequestCancellation	Cancellation process is triggered

5. IMPLEMENTATION:

5.1 High level description

The application was created to give a solution for storing passwords in a secure manner and with enhanced security features provided by the TEE environment. TEE's Trusted storage is the perfect environment to store sensitive data such as passwords.
The front-end application is a command line tool that is used to interact with the TA, via the open-tee engine that is build on top of a Linux-based system. However, it can be modified to attain interoperability and ne deployed to any other platform.

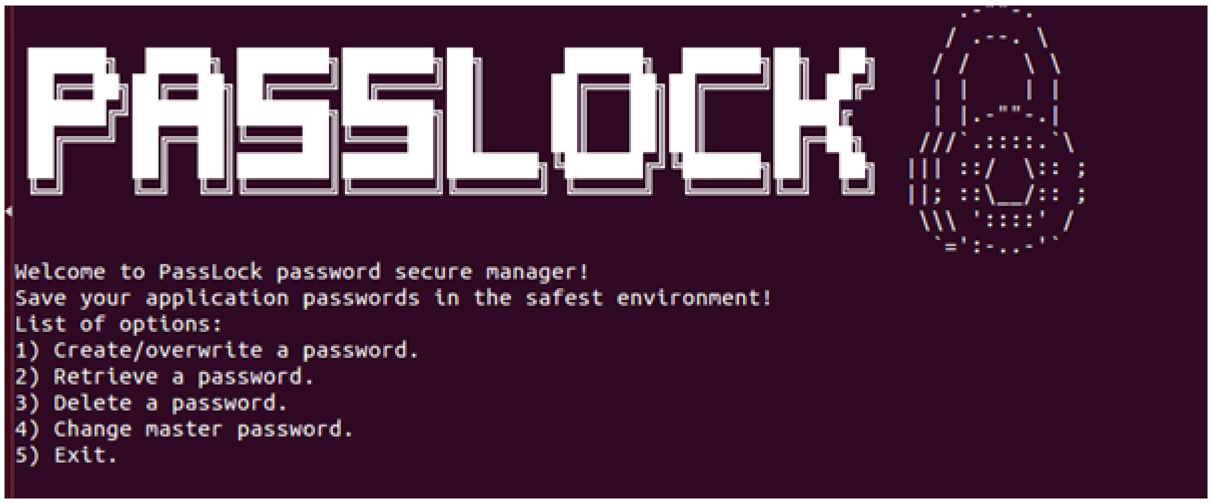


```
PASSLOCK

Give password:
```

Figure 5.1-6 Login Screen

The program operates as a simple password manager, it can store a password along with a username/index for it. To get access to the main program functionality though and to operate in the TA, one must provide the master password that is set the first time the application executes. The master password is securely encrypted and finally stored in the TA storage.



```
Welcome to PassLock password secure manager!
Save your application passwords in the safest environment!
List of options:
1) Create/overwrite a password.
2) Retrieve a password.
3) Delete a password.
4) Change master password.
5) Exit.
```

Figure 5.1-7 Main menu

It is noted that the application exits after a failed authentication attempt.
As mentioned earlier, the application uses the trusted execution environment secure storage to save the user’s passwords. It does so by sending the data over the channel that is setup upon applications' start, that is nothing more than a part of temporarily shared memory between the two applications.

5.2 Open Tee engine

The application is implemented on top of open-tee platform. Open-Tee engine is an emulator of TEE environment compliant with the GlobalPlatform TEE specifications. It is useful for developers and researchers that want to engage with tee development because as an emulator it is hardware independent. The engine implements all the APIs described in GlobalPlatform TEE specifications documents that provide all the functionality of a TEE and its underlying concepts.

5.3 Architecture

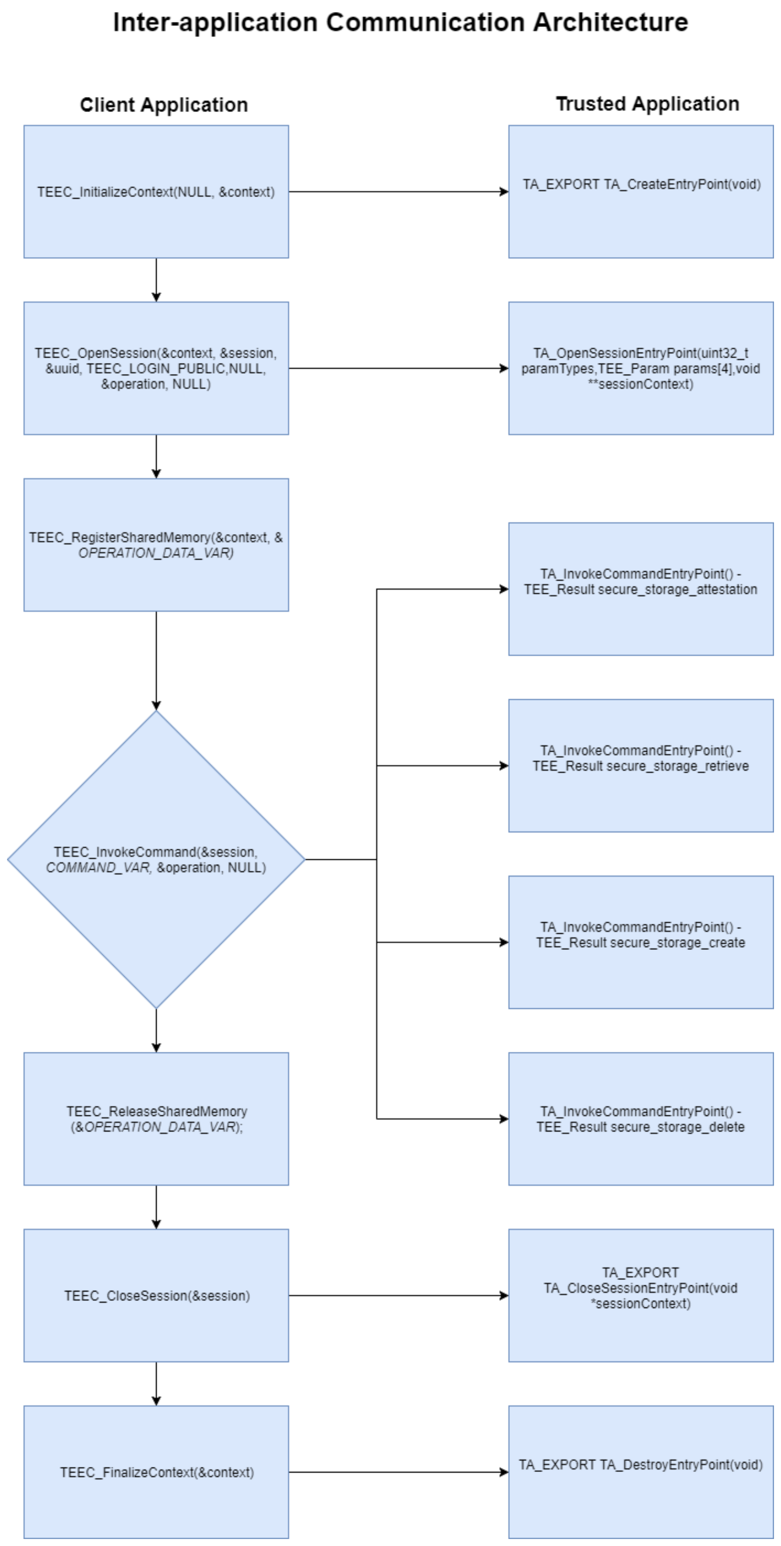

Inter-application Communication Architecture
Client Application
Trusted Application
TEEC_InitializeContext(NULL, &context)
TA_EXPORT TA_CreateEntryPoint(void)
TEEC_OpenSession(&context, &session, &uuid, TEEC_LOGIN_PUBLIC,NULL, &operation, NULL)
TA_OpenSessionEntryPoint(uint32_t paramTypes,TEE_Param params[4],void **sessionContext)
TEEC_RegisterSharedMemory(&context, & OPERATION_DATA_VAR)
TA_InvokeCommandEntryPoint() - TEE_Result secure_storage_attestation
TA_InvokeCommandEntryPoint() - TEE_Result secure_storage_retrieve
TEEC_InvokeCommand(&session, COMMAND_VAR, &operation, NULL)
TA_InvokeCommandEntryPoint() - TEE_Result secure_storage_create
TEEC_ReleaseSharedMemory (&OPERATION_DATA_VAR);
TA_InvokeCommandEntryPoint() - TEE_Result secure_storage_delete
TEEC_CloseSession(&session)
TA_EXPORT TA_CloseSessionEntryPoint(void *sessionContext)
TEEC_FinalizeContext(&context)
TA_EXPORT TA_DestroyEntryPoint(void)

Figure 5.3-8 Inter-application Communication Architecture

5.3.1 Trusted Application Architecture

The Trusted Application (TA) includes the SET_TA_PROPERTIES struct that contains the uuid that matches the TEEC_UUID unique id of the CA.

```
SET_TA_PROPERTIES(
   { 0x12345678, 0x8765, 0x4321, {'P','A','S','S','L','O','C','K'} }, /* UUID */
)
```

The architecture of the TA is simple and is strictly correlated with the CA functionality. As described earlier in this paper each client function/api call is triggering a function to the corresponding TA via the tee engine. That is to say whenever the CA initializes the context to openTEE engine, the TA_CreateEntryPoint function is invoked within the TA and it is the first function that is executed each time the application starts. It is also important to note that the TA_DestroyEntrypoint function is initialized alongside with the TA_ CreateEntryPoint function and it is used to finalize the application in the context of the opentee.

```
TEE_Result TA_EXPORT TA_CreateEntryPoint(void)
{
  OT_LOG(LOG_ERR, "Calling the create entry point");
   return TEE_SUCCESS;
}
void TA_EXPORT TA_DestroyEntryPoint(void){
   OT_LOG(LOG_ERR, "Calling the Destroy entry point");
}
```

Respectively whenever the TEEC_OpenSession() is called from the CA, the TA_OpenSessionEntryPoint() function is called from within the TA and the TA_CloseSessionEntryPoint() is used to close the session between the two applications. After a session has been established between the two applications the TA is "listening" for incoming commands from the CA. For this purpose four constants are declared at th start of the TA that represent the unique id related to each command. The commands are the following:

```
#define CREATE   0x00000001
#define RETRIEVE  0x00000002
#define DELETE     0x00000003
```

```
#define ATTESTATION 0x00000004
```

Each command represents a function implemented in the TA that is called every time the corresponding id is given.

CREATE command is used for password creation operation or even for overwriting a password entry.

RETRIEVE command is used to retrieve the password that corresponds to the given id (source CA).

DELETE command deletes the password that corresponds to the given id (source CA).

ATTESTATION command is used to compare the stored hash to the given hash from the CA so as to perform a preliminary validation against the CA.

```
TEE_Result result = TEE_SUCCESS;
   switch (cmd_id){
      case CREATE:
         result=secure_storage_delete(param_types, params);
         result= secure_storage_create(param_types, params,sess_ctx);
         return result;
         break;
      case RETRIEVE:
         return  secure_storage_retrieve(param_types, params,sess_ctx);
         break;
      case DELETE:
         return secure_storage_delete(param_types, params);
         break;
      case ATTESTATION:
         return secure_storage_attestation(param_types, params);
         break;
      default:
         return TEE_ERROR_BAD_PARAMETERS;
```

The CREATE and RETRIEVE command is also used for creating/changing the master password and also for authentication with it. Thus there are more parameters given to the corresponding functions, so as to decide which operation to execute and also there is a check if the master password exists in the TA memory or else the application is automatically redirected to the master password creation function.

The need for usage of persistent objects in the TA memory arises due to three reasons. Firstly the hash regarding the attestation process must be permanently saved in the TA and also the hash of the master password (once it is created) is needed to remain in memory, until the CA calls the CREATE function with a parameter to change the master password. It is obvious that the password entries need also to be stored in the memory until deleted or overwritten.

For this reason, two constants ids are initialized at the start of the program:

```
#define PASS_ID 0
#define HASH_ATT_ID 1
```

These constants are references to the master password id and the attestation hash in memory.

TEE_Result secure_storage_create function:

As mentioned earlier the create command is used to create password entries as well as the master password. Whenever the TEE_Result secure_storage_create is called the function checks its calling parameters to identify if it’s a master password creation call or a simple password entry creation one. After this it creates the proper persistent object in memory:

```
TEE_ObjectHandle persistent_object = (TEE_ObjectHandle)NULL;\
Result=TEE_CreatePersistentObject(TEE_STORAGE_PRIVATE,&objectID,objectIDsize,flags_create,N
ULL,params[1].memref.buffer, params[1].memref.size,&persistent_object);
```

This object is used to store the data given to it upon creation and it has also a unique id so as to be referenced by the application.

***The password entries are overwritten if the call is used with an existent object id.*

TEE_Result secure_storage_retrieve function:

TEE_Result secure_storage_retrieve is mainly used to retrieve the password entries, but also for user authentication. That is to say when the user types the master password a TEE_Result secure_storage_retrieve function call is performed to check the master password hash in the TA memory.

```
/*checking existence of master pass*/
    if(param_types ==
TEE_PARAM_TYPES(TEE_PARAM_TYPE_NONE,TEE_PARAM_TYPE_NONE,TEE_PARAM_TYPE_N
ONE,TEE_PARAM_TYPE_NONE)){
        uint32_t objectID=PASS_ID;
        uint32_t objectIDsize = sizeof(objectID);
        result = TEE_OpenPersistentObject(TEE_STORAGE_PRIVATE,&objectID,
objectIDsize,flags_open,&persistent_object);
        TEE_CloseObject(persistent_object);
    }
```

TEE_Result secure_storage_delete function:

This function is used to delete password entries, by password id. It is also used to delete the master password (depending on the parameters given to it). The passwords are deleted using the TEE_CloseAndDeletePersistentObject function of the internal api that is used to delete persistent objects in TA memory.

```
result=TEE_OpenPersistentObject(TEE_STORAGE_PRIVATE,&objectID,
objectIDsize,flags_delete,&persistent_object);
        TEE_CloseAndDeletePersistentObject(persistent_object);
    }
```

TEE_Result secure_storage_attestation:

The attestation related function is called to compare the hash given from the CA to the one stored in TA memory and it is doing so with the persistent_object handling functions mentioned above.

6. CONCLUSION

This thesis deals with the development of a Trusted Application in the context of open-tee engine. The application is a password manager that uses all the security features that are provided by a platform such as open-tee, making it a more secure solution to common password managers.

Open-TEE is a helpful tool for developers that want to start creating applications using a TEE environment, as it removes the need of costly hardware components and takes care for all the TEE engine functionality.

As a result, little knowledge of Hardware programming is needed to develop a complex application on top of TEE and also the debugging process is far more simple and fast in this virtual environment than an actual hardware TEE.

What is needed to be able to develop an application within open-tee environment, is a strong understanding of its main functionality and core architecture, along with all its APIs that implement the TEE behavior.

The security standard that such a solution provides is far greater than the ordinary applications. TEE has many security features that can be used for applications that handle sensitive data. The application, developed during this thesis, is a strong a secure password manager at this point, however it does not make use of every one of those features and so it can be enhanced and improved in the future, providing more usability and security.

7. BIBLIOGRAPHY

1. https://www.gsma.com/digitalcommerce/mobile-security-technology-overview-trusted-execution-environment-tee
2. https://www.embedded.com/design/connectivity/4430486/Trusted-execution-environments-on-mobile-devices
3. http://ieeexplore.ieee.org/document/6799152/?reload=true
4. https://www.researchgate.net/publication/262412456_Trusted_execution_environments_on_mobile_devices
5. GlobalPlatform, " TEE System Architecture," http://www.globalplatform.org/specificationsdevice.asp.
6. GlobalPlatform, "Device specifications for trusted execution environment." http://www.globalplatform.org/specificationsdevice.asp.
7. The Chromium Projects, "Multi-process architecture," http://www.chromium.org/developers/design-documents/multi-process-architecture. Muthu, "Emulating trust zone feature in android emulator by extending qemu," Master's thesis, KTH Royal Institute of Technology, 2013.
8. https://www.electronicsweekly.com/blogs/eyes-on-android/what-is/what-is-a-trusted-execution-environment-tee-2015-07/
9. The Apache Software Foundation, "Apache license, version 2.0," http://www.apache.org/licenses/LICENSE-2.0.
10. https://arxiv.org/pdf/1506.07367.pdf
11. https://arxiv.org/pdf/1506.07739.pdf
12. https://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/23823/Dettenborn.pdf?sequence=1
13. https://www.cs.helsinki.fi/group/secures/CCS-tutorial/tutorial-proposal.pdf
14. http://www.ibm.com/security/cryptocards/
15. J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for TCB minimization," in Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008, 2008, pp.315–328. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352625
16. Linaro, " OP-TEE,"https://wiki.linaro.org/WorkingGroups/Security/ OP-TEE.