ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
UNIVERSITY OF PIRAEUS

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

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

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΘΕΜΑ

¨ TRUSTED EXECUTION ENVIRONMENT ¨

Σπουδαστής:

ΣΟΦΙΟΣ ΙΩΑΝΝΗΣ ΜΤΕ 1535

Επιβλέπων Καθηγητής:

Δρ. ΞΕΝΑΚΗΣ ΧΡΗΣΤΟΣ

ΙΟΥΝΙΟΣ 2017

ABSTRACT

As Mc Gillion, Dettenborn, & Nyman described[1], hardware-based Trusted Execution Environments (TEEs) are widely deployed in mobile devices. Yet their use has been limited primarily to applications developed by the device vendors. Recent standardization of TEE interfaces by GlobalPlatform (GP) promises to partially address this problem by enabling GP-compliant trusted applications to run on TEEs from different vendors. Nevertheless ordinary developers wishing to develop trusted applications face significant challenges. Access to hardware TEE interfaces are difficult to obtain without support from vendors. Tools and software needed to develop and debug trusted applications may be expensive or non-existent. Open-TEE conforms to GP specifications. It allows developers to develop and debug trusted applications with the same tools they use for developing software in general. Once a trusted application is fully debugged, it can be compiled for any actual hardware TEE. Through performance measurements and a user study we demonstrate that Open-TEE is efficient and easy to use. We have made OpenTEE freely available as open source

ΕΥΧΑΡΙΣΤΙΕΣ

Η παρούσα διπλωματική εργασία διενεργείθηκε στα πλαίσια του προγράμματος παρακολούθησης του ΠΜΣ Τεχνοικοινομικής Διοίκησης και Ασφάλειας Ψηφιακών Συστημάτων(2015-16) του Πανεπιστημίου Πειραιά.

Αισθάνομαι την υποχρέωση να ευχαριστήσω όλους τους καθηγητές του μεταπτυχιακού προγράμματος οι οποίοι, χάριν στα πλούσια πνευματικά τους προσόντα και τις πολύτιμες γνώσεις τους, συνέβαλαν στην ολοκλήρωση των μεταπτυχιακών μου σπουδών.

Ιδιαίτερα αισθάνομαι την ανάγκη να ευχαριστήσω τον επιβλέπωντα καθήγητή μου Δρ. Ξενάκη Χρήστο που με την καθοδήγηση του, τον επαγγελματισμό του, την εμπιστοσύνη που έδειξε στο πρόσωπο μου αλλά και το ήθος του, κατάφερε να διεγείρει το ενδιαφέρον μου για περαιτέρω έρευνα και γνώση στις επιστήμες της πληροφορικής συμβάλλοντας ουσιαστικά στην βελτίωση της επαγγελματικής μου πορείας.

Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου που για ακόμη μια φορά με στήριξαν και στάθηκαν αρωγοί δίπλα μου, σε αυτήν μου την επιλογή για περαιτέρω γνώση και επιμόρφωση.

Table of Contents

LIST OF ABBREVIATIONS

AMD-V	AMD Virtualization
API	Application Programming Interface
CA	Client Application
CI	Continuous Integration
COW	Copy-On-Write
CPU	Central Processing Unit
DRM	Digital Rights Management
eMMC	embedded Multi Media Card
GP	Global Platform
GPL	General Public License
HMAC	keyed-Hash Message Authentication Code
HSM	Hardware Security Module
HW	Hardware
ICRI-SC	Intel Collaborative Research Institute for Secure Computing
IDE	Integrated Development Environment
Intel VT-(x,d)	Intel Virtualization
IOMMU	Input/ Output Memory Management Unit
iOS	iPhone OS
IPC	Inter-Process Communication
JNI	Java Native Interface
JTAG	Joint Test Action Group
LoC	Lines of Code
NVM	Non-Volatile Memory
OS	Operating System
ObC	On-board Credentials
ODM	Original Design Manufacturer
OEM	Original Equipment Manufacturer
Open-TEE	Open Virtual Trusted Execution Environment
OS X	Mac OS X
PC	Personal Computers
PKCS	Public-Key Cryptography Standards
PSS	Proportional Set Size
RAM	Random Access Memory
REE	Rich Execution Environment
ROM	Read Only Memory
RPC	Remote Procedure Call
RPMB	Replay Protected Media Block
RSS	Resident Set Size
SDK	Software Development Kit
SGX	Intel Software Guard Extensions

SMC	Secure Monitor Call
SoC	System on Chip
SUS	System Usability Scale
TA	Trusted Application
TCPA	Trusted Computing Platform Alliance
TEE	Trusted Execution Environment
TLK	Trusted Little Kernel
TLS	Transport Layer Security
TPM	Trusted Platform Module
TUI	Trusted User Interface
USB	Universal Serial Bus
UUID	Universally Unique Identifier
VMM	Virtual Machine Manager

1. Introduction

1.1 TEE

Devices, from smartphones to servers, offer a Rich Execution Environment (REE), providing a hugely extensive and versatile operating environment. This brings flexibility and capability, but leaves the device vulnerable to a wide range of security threats. The Trusted Execution Environment (TEE) is designed to reside alongside the REE and provide a safe area of the device to protect assets and execute trusted code.
This document explains the hardware and software architectures behind the TEE. It introduces TEE management and explains concepts relevant to TEE functional availability in a device.
At the highest level, a Trusted Execution Environment (TEE) is an environment where the following are true:

- Any code executing inside the TEE is trusted in authenticity and integrity.
- Other assets inside the TEE are trusted in authenticity and integrity and protected in confidentiality.
- The TEE resists known remote and software attacks, and a set of external hardware attacks.
- Both code and other assets are protected from unauthorized tracing and control through debug and test features.

The architectural concepts and principles in this document do not and should not dictate any particular hardware or software implementation and are broad enough to cover many possible implementations as long as the security principles are adhered to. Hence, any hardware or software architectural diagram in this document should be taken as an example and for reference only.
This version of the TEE System Architecture has been extended to include the second phase of TEE standardization which introduced new APIs for supporting tasks such as Trusted User interface, SE and Sockets communications, and remote management for Trusted Applications. Further extensions of the TEE System Architecture are expected in subsequent phases, as described in the TEE White Paper [TEE White Paper]; e.g. a more flexible Trusted User Interface API, biometrics fingerprint API, secure video content.

Smart connected devices, such as smartphones, are intrinsic to daily life: they are used for business, social interactions, making purchases and enjoying media content. All of this data, however, is susceptible to attacks from hackers and the millions of downloadable applications represent an even larger opportunity for fraudsters.

Similarly, automotive and home devices are increasingly becoming connected and offering more functionality. On top of this, consumers are increasingly using

their devices in new ways: organizing a trip from a smart TV, streaming music while driving or using a smartphone to pay for shopping. These expanded practices create new security vulnerabilities, which highlight the need for mechanisms that allow trusted parties to have access to applications without granting hackers the same opportunity.

Service providers and original equipment manufacturers (OEMs) now need to protect applications on many levels: from attacks originating in a device's operating system, authenticating the correct user to the correct service, offering increased privacy, protecting valuable content, allowing secure access to corporate and personal data and mitigating financial risks. One solution to these security challenges is to provide a small, isolated execution environment that allows service providers and OEMs to improve the user experience while reducing fraud. The GlobalPlatform Trusted Execution Environment (TEE) effectively addresses these concerns.

A TEE is a secure, integrity-protected processing environment, consisting of processing, memory and storage capabilities. Figure 2.1 shows how a device can be visualized as a series of distinct environments with their own set of features and services. What follows, using the terminology introduced by GlobalPlatform [18], describes the concepts illustrated in Figure 2.11.

1.1.1 Rich Execution Environment (REE)

The word "rich" here refers to an operating environment that is feature rich, as one would expect from modern platforms such as Android, iOS, Windows, Linux or OS X. In some literature this environment may be referred to as the "Normal World" in reference to the fact that it is where the majority of applications are being developed for and deployed to.

1.1.2 Trusted Execution Environment (TEE)

The TEE is a combination of features, both software and hardware, that isolate the execution of tasks from the REE. These environments have a limited set of features and services as they are intended to only address the security critical subset of an application's functionality such as offloading some cryptographic operations or key 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 TEE definition

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.

1.3 Creation of TEE

Multiple handset and chip manufacturers have already developed and deployed proprietary versions of this technology. The resulting lack of standardization 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.

There are two central reasons why the TEE exists:

- **An increasing number of mobile services, which require a greater level of security, are emerging.**
- **With a growing number of users, there is a greater need for protection against software attacks**. Applications with higher security requirements, and therefore heightened ramifications if compromised, require more protection than can be offered by rich OS solutions alone.

Enterprise IT environments, delivery of premium multimedia content, mobile payments, the Internet of Things, government identification programs and more seek to balance a consumer's desire for a rich experience with the security concerns shared by consumers and service providers. The TEE isolates trusted applications and keeps them away from any malware which might be downloaded inadvertently. Because of this, the TEE will become an essential environment within all devices as the secure services market evolves.

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.

1.4 Use of TEE

Utilizing a TEE is not a silver bullet for securing a device. It provides defense in depth and helps narrow down the attack vectors that an attacker can leverageto compromise a device or user. Properly offloading tasks to TEE can efficiently protect sensitive data from leaking, however, if we assume the following scenario: a user has downloaded a new update for their device which contains malicious code. If that malicious application can masquerade as a legitimate CA, then the attacker could have free use of the sensitive data stored in the TEE. That is they would be able to e.g. decrypt data or sign messages as a legitimate user, however, the TEE would still ensure that the key was not revealed. Even with this limitation the benefits of using a TEE far outweigh the risks of not using it

and a TEE is critical to the proper functioning of certain use cases that have become commonly available in mobile devices, such as:

Keystore: Used for storing cryptographic tokens, keys or certificates, into a TEE

to make it more difficult to extract them from the device. When tokens are deployed to a TEE, a CA can make use of them through a keystore Application Programming Interface (API) and these tokens are thus not exposed to user space or Random Access Memory (RAM).

Secure storage: This can serve as a multi-purpose facility which could allow an

application to store everyday information such as user identity, pictures or documents. Most implementations provide both confidentiality and integrity protections. In fact modern storage media such as embedded MultiMediaCard (eMMC) may contain a special partition, called the Replay Protected Media Block (RPMB), to assist with integrity protection. It relies on a keyed-Hash Message Authentication Code (HMAC) for its operation and this key is protected by the TEE. Mobile phone vendors have used secure storage for calibration data and firmware configurations for many years to protect their assets and the safety of end users. For example an attacker, in this case the legitimate owner of a device, may wish to alter the modem configuration to allow them to have a greater share of the bandwidth or a higher priority on the network. This can lead to poor service for other users or be potentially harmful to the user as the new settings may pose a health risk. In order to mitigate this risk the manufacturer would store these critical configurations in secure storage and would potentially disallow the device to boot if the calibration data has been tampered with.

Secure boot: is an extension to what has just been discussed, it provides the ability to measure the integrity of certain code and data during the boot and can be designed to disallow boot if any of the components have been altered. It does this by storing a list of known good configurations, i.e. the signatures of firmware and software components and comparing these to the components as they are prepared for loading. Modern implementations of secure boot extend from the hardware all the way to the user space. Non-Volatile Memory (NVM) such as the Read Only Memory (ROM) code or key hashes stored in physical write once fuses can be used to provide the basis of security. The systems can be thus designed that each verified component can be relied upon to provide the verification of the layer(s) above it. A simplified example would be that the ROM code verifies the bootloader, which in turn verifies the main OS, which in turn verifies the overall REE before launching the first user space application e.g. init. Secure boot does

not guarantee that the device is free of security issues; rather it can certify that the components that have booted are the best known configuration as provided by a trusted source, e.g. the OEM.

Digital Rights Managements (DRM): One of the driving forces behind the wider adoption of TEEs has been the media industry. DRM content is becoming ubiquitous, common examples are music files from iTunes2 or a video stream from HBO3. In order to protect the media stream from piracy the data is encrypted with a key that is generally device or session specific. The TEE is used to protect this key, perform the decryption of the session and make the data available to other parts of the system so it can be securely displayed.

Although the previous use cases are more prevalent on mobile devices they are now seeing widespread deployment on a variety of devices ranging from desktops to smart watches. This thesis hopes to outline that although these are some of the most common uses of a TEE, it is by no means an exhaustive list and in fact TEE technology is underutilized [14].

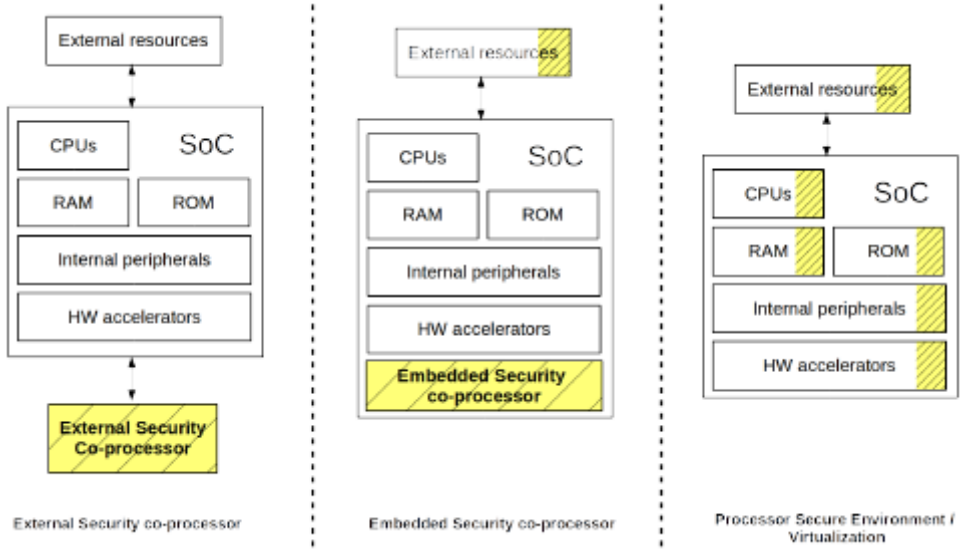


Figure 1.1

There are three main use cases for the TEE. It can be used to protect:

1. Digital content such as films, television, music and other multimedia formats,
2. mCommerce and mPayments credentials and transactions,

3. Enterprise and government data.

The protection of premium content, such as a 4K resolution film or a TV series which has just been aired, is a key driver for the adoption of TEE technology. TEE technology can be used to ensure that content cannot be stolen once it is decrypted on a device. It does this by offering a trusted environment in which to perform the decryption and store the file, in addition to offering trusted video playback to protect the content while it is being displayed on screen. The technology is therefore of great value for smartphones and tablets, in addition to 4K televisions and set top boxes.

In mCommerce and mPayments, TEE technology is already being used to protect payment credentials such as cryptographic keys while a transaction is being authorized. Another benefit of the TEE is the ability to offer a **trusted user interface (UI)** which ensures that the correct information is displayed to the user and that the information displayed on screen and entered by the user is secure. These capabilities reduce the risk of passcode logging and allow transaction, logs and statement information to be securely displayed.

In an enterprise or government environment, the protection of corporate or otherwise sensitive data is essential. Bring your own device (BYOD) is becoming ever more prevalent as more employees use their own handsets and tablets to perform work- based tasks like email and document editing. The TEE enables the secure handling of confidential data, protection against software attacks from the Rich OS and assistance with access rights control and user authentication.

1.5 Benefits of 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 in order 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.

1.6 TEE & Smartphone infrastructure

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.

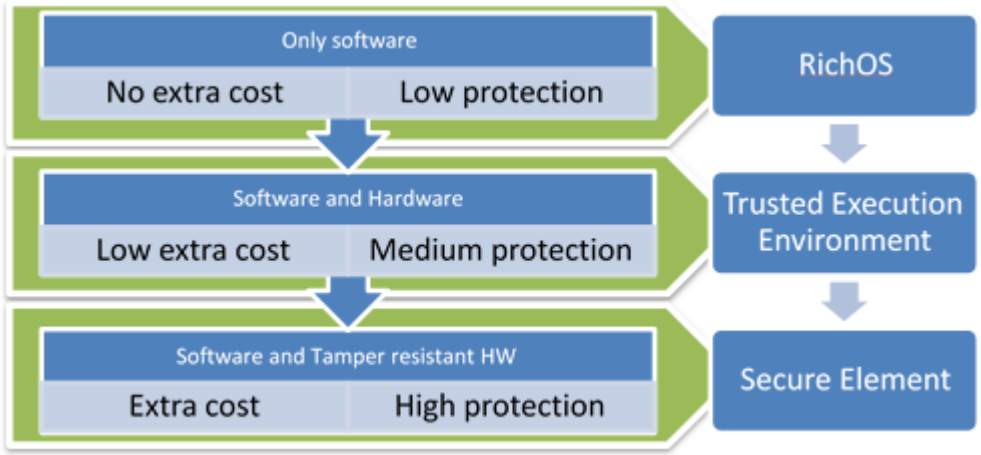


Figure 1.2

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).

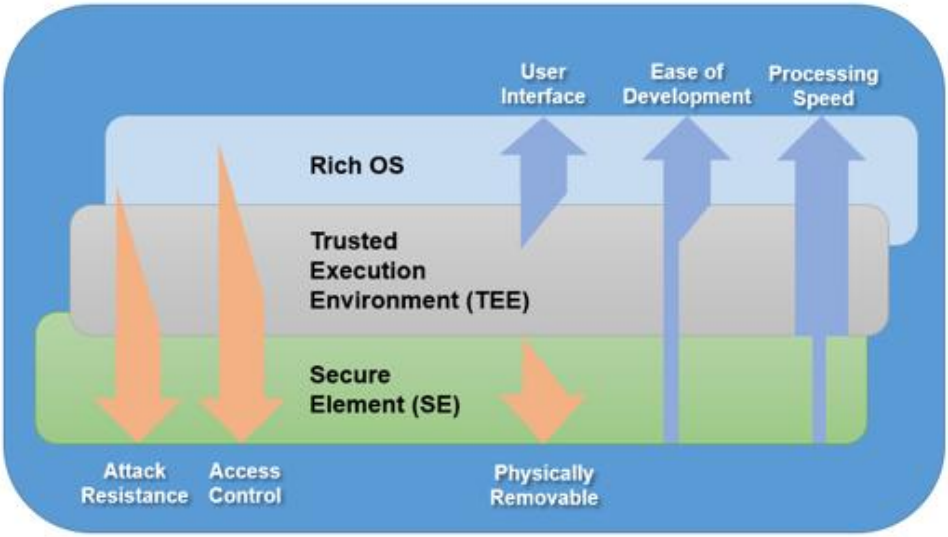


Figure 1.3

1.7 TEE Standardization

TEE standardization is essential to avoid fragmentation. The proliferation of proprietary TEE solutions would lead to the following:

- Higher costs to develop or change applications/solutions when creating or adapting to proprietary platforms
- The need for very specialized skills
- Extended time-to-market due to longer development times and potential integration issues.

Standardization, by contrast, enables simplified and unified implementation and improves interoperability between stakeholders. Furthermore, standardization allows a large ecosystem to thrive and blossom, allowing for multiple business partners and, because it ensures long-term stability and survivability, protects investment in a way that proprietary solutions cannot. It also defines a basis for evaluating and comparing different solutions. Lastly, standardization creates a foundation for a uniform certification process.

The landscape for TEEs has been very diverse, with a variety of different architectural options from multiple manufacturers. Even platforms using the same type of TEE are often not interoperable. For example, an application written for one TrustZone-based platform will generally not run on a different TrustZone-based platform. They may be using different TEE OSs or different REE OS drivers. On the other hand, developers and others who are higher up in the software ecosystem are less concerned with intricacies of low-level software or hardware but more concerned with their ability to use the capabilities of TEEs easily and across different platforms. This calls for standardization.

1.7.1 GlobalPlatform

One initiative in TEE standardization has been undertaken by GlobalPlatform [17], which "is a cross industry, non-profit association which identifies, develops and publishes specifications that promote the secure and inter-operable deployment and management of multiple applications on secure chip technology" [15]. GP offers specifications in three areas: smartcards, back-end support systems and devices. This thesis is concerned with specifications from the device working group related to the APIs for TAs. Figure 2.3 shows the primary interfaces standardized by GP. The GP TEE Core API provides an extensive set of features such as a crypto API and secure storage that can be used to implement a TA, for example a DRM decoder. The GP TEE Client API is a very generic and thin layer consisting of a small number of functions and definitions that allow the transfer of data back and forth from the REE to a TA. A CA, for example a DRM player, will implement all complex but non-critical functionality by itself, but use the GP TEE Client API to invoke the corresponding TA, such as the DRM decoder. Between the "GP TEE Client API" running on the REE and "GP TEE core API" running on the TEE we have an effective Remote Procedure Call (RPC) mechanism where a process running in the REE can invoke tasks in the TEE.

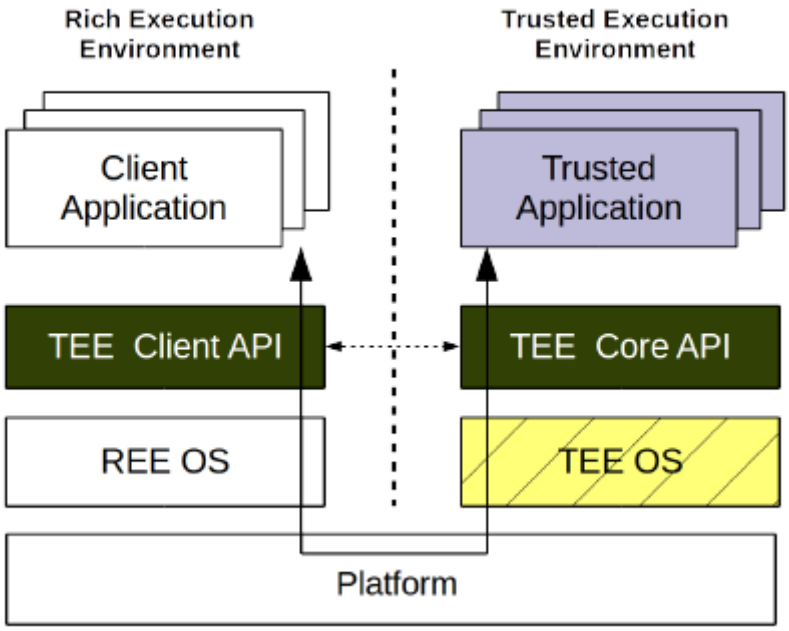


Figure 1.4

1.7.2 Benefits of TEE Standardization

These standardization efforts in GlobalPlatform could resolve the issue of interoperable TEEs. In other words, TEE application developers could re-use the same application across different TEEs rather than developing for a specific TEE. For example, a keystore service is already provided by Intel SEP and ARM TrustZone, but unfortunately both define their own APIs, which forces the implementation of tailored solutions on each platform.

If the TEE vendors were to agree on requirements and standards, in addition to committing to deploy and use them in their products it may provide an incentive for the developer community to utilize these features. A wider research and developer community could be a valuable resource as they will help to drive the next revolution in TEE use cases. It is already abundantly clear that there is a consumer need for more security, privacy and identity protection being among their top concerns so we can expect more demand for TEE functionality to come. Even taking existing standards and enabling them, such as implementing the Public-Key Cryptography Standard #11 (PKCS#11), in a readily accessible way can have a large immediate impact. PKCS#11 defines a generic interface to cryptographic tokens e.g. how to sign data. It is a well-known and used standard though it generally relies on 3rd party hardware, such as smartcards, smartcard readers, Universal Serial Bus (USB) dongles etc. Numerous languages provide wrappers for the API, many more applications are PKCS#11 aware 5 and can be configured to offload cryptographic operations to a PKCS#11 implementation. Providing this, integrated, as part of an existing device will free the end user from having to know which hardware to carry around with them and will enable application developers to tailor their applications knowing that this feature is readily available.

However, given all the benefits that standardization brings it does not remove the obstacle of gaining access to the requisite hardware nor does it simplify the task of developing and testing TAs. The remainder of this thesis focuses on how to overcome this lack of access and enable developers and researchers to gain valuable experience with TEE technology and especially its concepts.

1.7.3 GlobalPlatform achievements

GlobalPlatform's 120+ members recognize the need for standards to be developed in parallel with the evolution of a new ecosystem. This mutual

development will provide greater certainty and lower the cost of progress for the industry by removing barriers caused by a lack of interoperability.

Specifications : With 17 years of experience in the mobile space and the expertise of a global membership which represents the full ecosystem, GlobalPlatform's work is leading the market. GlobalPlatform Card Specifications are now embedded in more than 17.7 billion SEs. Since the TEE Client API v1.0 was published in July 2010, GlobalPlatform has been responsible for driving TEE standardization on behalf of the industry. Since that time, the following specifications have been developed / delivered by GlobalPlatform:

1. TEE Client API Specification v1.0 – enables communication between applications running in a Rich OS and trusted applications residing in the TEE.
2. TEE Internal Core API Specification v1.1.1 – enables trusted applications within a TEE to perform the general operations of a security application, such as cryptography, secure storage, communication and general tasks, such as timekeeping and memory management.
3. TEE Secure Element API Specification v1.1 – allows trusted applications to directly communicate with a SE, rather than through a client application.
4. TEE Sockets API Specification v1.0 – is a suite of specifications that provide standards to enables trusted applications to directly make use of internet protocol interfaces, rather than send packets to a client application for internet transfer.
5. Trusted User Interface API Specification v1.0 – allows a trusted application to securely display text and graphics, and ask the user to perform an action ranging from navigation to entry of an associated PIN- or Password-backed ID.
6. TEE Systems Architecture v1.0 – explains the hardware and software architectures behind the TEE.
7. TEE Internal API Specification v1.0 – specifies how to develop trusted applications.
8. TEE Protection Profile v1.2 – facilitates the Common Criteria evaluation of TEEs.
9. TEE TA Debug Specification v1.0.1 – enables the debugging of GlobalPlatform compliant TEEs.
10. TEE Compliance Profile - combines the functional testing of the TEE Client API and the TEE internal core API.
11. Secure Element Remote Application Management v1.0.1 – defines a single administration protocol to perform remote management of applications residing on any type of SE.
12. Secure Element Access Control v1.1 – specifies how the access policy is stored in the SE and how it can be accessed and applied by the device.

All specifications can be downloaded from the GlobalPlatform Device Specifications webpage.

The GlobalPlatform Compliance Program : To promote confidence within this advancing ecosystem, GlobalPlatform has launched a TEE compliance program. This offers assurances to application and software developers and hardware manufacturers that a TEE product will perform in line with the GlobalPlatform specifications and as intended. It also promotes market stability by providing a long-term, interoperable and industry agreed framework that will evolve with technical requirements over time. Visit the GlobalPlatform Compliance Program webpages for further information.

Security certification : To complete this infrastructure, in February 2015, GlobalPlatform's TEE Protection Profile was officially certified by Common Criteria. Product vendors are now able to undertake a formal security evaluation of their TEE products, using laboratories licensed by supporting certification bodies to evaluate and certify that they meet the security requirements in the document.

GlobalPlatform has also launched a TEE Certification Scheme that evaluates the security level of a given TEE implementation. To drive this initiative, GlobalPlatform has also launched a TEE Security Evaluation Secretariat to manage the scheme. Under the scheme, providers of TEE products will be able to submit their products to the new GlobalPlatform secretariat for independent evaluation of their conformance to the organization's TEE Protection Profile.

In the mid-term, GlobalPlatform is working to accelerate the deployment of certified TEEs and to create an ecosystem where GlobalPlatform certification is a prerequisite amongst service providers and handset manufacturers. This is a stepping stone on the way to achieving full market adoption, with the long-term goal of the specifications becoming a de facto standard for the industry.

2. TEE Architecture

2.1 View of TEE

A TEE can be realized in different ways, but the overall concept stays the same.

Figure 2.2 shows a number of ways in which these TEEs can be realized:

2.1.1 Co-Processor

A separate core, generally with its own peripherals, is used to offload the security critical tasks from the main operating environment. The benefits of such a configuration are that the operation can generally be completely isolated and it can run simultaneously with the main core. The drawback is that there is an overhead associated with transferring the data to and from the core. Also, the co-processor is generally less powerful than the main core. The co-processor design can be further separated into two alternatives:

External security co-processor is a discrete hardware module outside the physical chip (commonly referred to as "System on Chip" or SoC) containing the main core, and is thus completely isolated from it, not sharing any resources with it.

Embedded security co-processor is embedded into the main SoC and thus has the capability to share some of the resources of the main system. It is still isolated from the main processor.

2.1.2 Processor Secure Environment

Many popular mobile TEE architectures follow a configuration where a single core supports multiple virtual cores that are mutually exclusive of one another i.e. when one is running the other is suspended. Generally there is some form of trigger to allow the core to switch from one state to the other. This configuration is sometimes referred to as the "processor secure environment" [12].

ARM TrustZone is an example of this configuration. In TrustZone, the processor core can be in one of two "worlds": a "secure world" (for the TEE) and a "normal world" (for the REE). A special instruction called Secure Monitor Call (SMC) can be executed to trigger the processor running in normal world to enter "monitor mode" that marshals the transition to secure world [4]. The advantage of this configuration is that there is no need to offload the data to and from the secure world. However, there is a cost associated with having to store and restore the device state on entry and exit from a given mode. On single core devices there is also an added security benefit from having only one world running at a given time in that it ensures that the normal world OS cannot interfere with the secure

world directly or indirectly (e.g., software side-channel attacks). However, this also has the disadvantage that when one world is active the other world must be completely halted, thus complicating interrupt handling and potentially causing a transition back to handle the interrupt before the task is complete.

Intel Software Guard Extensions (SGX) [29, 21] is another example of such

a variant, the core does not perform a full transition to and from a secure world.

Instead parts of a standard application, both code and data, are protected by mechanisms in the core. Parts of the application, called an Enclave, are encrypted by a key that is only accessible to the Central Processing Unit (CPU). When an "enter enclave (EENTER)" [23] instruction is received the code and data are decrypted and operated upon in the core. They never leave the CPU package unencrypted, thus protecting them against external access. The benefits are that there is no need to transfer data back and forth between cores or to setup complicated transitions to and from a secure world, and there is no additional need for a separate operating environment as is required in other styles of TEE configuration. Most security use cases related to normal world applications running can be supported in this fashion by allowing small select parts of the application to be secured, thus enabling developers to move away from the paradigm of "Splitting Trust" [6] towards a model that developers are more familiar with, where the application can be self-contained. Obviously this does not remove the need for the developer to know which parts of an application need to be secured and thus which parts to run in an enclave, however, it does remove the need to develop a separate application that will run within a different environment; which in the case of a co-processor may have a completely different set of tools and requirements for the developer to learn in order to utilize the facility.

2.1.3 Virtualization

Virtualization based on hardware features such as AMD Virtualization (AMD-V)

and Intel Virtualization (Intel VT-[x,d]) have existed for many years and are used

extensively to provide separation of resources between different operating environments especially in high density server configurations. They rely on processor support to allow virtualization of instructions and access to resources e.g. through the use of an IOMMU (Input/Output Memory Management Unit) access to and from peripheral devices can be restricted. Though in and of

themselves they are not designed solely to provide a TEE, there is recent research [9, 28] to see how these can be used as an alternative to dedicated hardware based TEEs. When deployed as TEE environments they generally rely on a Virtual Machine Manager (VMM) to provide the marshaling of access to the resources. There has been extensive research and security auditing of VMM technology in recent years 4, to the point where it is now uncommon to hear of exploits against the VMM itself. Vulnerabilities such as venom [10] highlight that there is still always the possibility of vulnerabilities in any sufficiently large code base, however, in contrast to traditional hardware based TEEs there is generally an open disclosure of the vulnerabilities and a software patch is, in many cases, a sufficient remedy.

2.2 TEE Device Architecture Overview

A TEE is an execution environment which provides security features such as isolated execution, integrity of Trusted Applications (TAs), and integrity and confidentiality of TA assets.
A GlobalPlatform TEE is defined as one that meets both the following criteria:

- GlobalPlatform security certification, that it must meet the security standard defined by the GlobalPlatform TEE Protection Profile [TEE PP].
    - If the TEE is claimed to fully support other GlobalPlatform TEE specifications, it must do so in a security certified manner.
    - It should be noted that this means the TEE must provide separation from other environments in the device (including other TEEs)
- GlobalPlatform functional qualification, that the TEE must support at least the initial TEE configuration, which currently consists of being compliant with:
    - GlobalPlatform TEE Client API Specification [TEE Client API]
    - GlobalPlatform TEE Internal Core API Specification [TEE Core API]
    - If the TEE is claimed to fully support other GlobalPlatform TEE specifications, it must do so in a functionally compliant manner.

For a particular device, proof of meeting the above criteria must be obtained from relevant and approved certification and compliance laboratories. More information can be found on the GlobalPlatform website.
Note:

- The presence of a GlobalPlatform TEE on a device does not restrict the presence of other Trusted Execution Environments that are not GlobalPlatform compliant.
- A GlobalPlatform TEE may have better security and/or more capabilities than those required by GlobalPlatform.

The remainder of this chapter describes the general device architecture associated with the TEE along with a high level overview of the security requirements of a TEE.
There is no mandated implementation architecture for these components and they are used here only as logical constructions within this document.

2.2.1 Typical Chipset Architecture

The board level chipset architecture of a typical mobile device is depicted in Figure 2-1. The chipset hardware consists of a Printed Circuit Board (PCB) that connects a number of components such as SoC processing units, RAM, flash, etc.

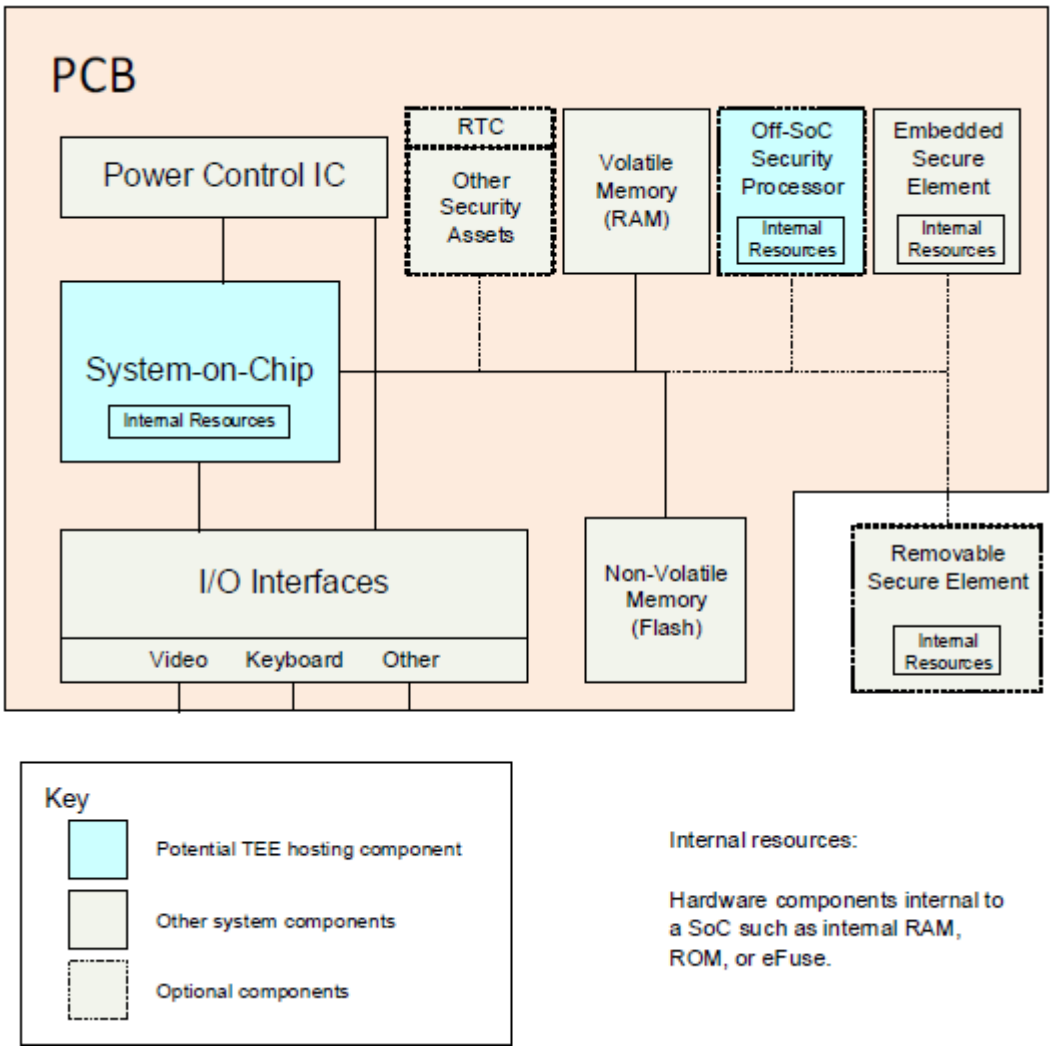


Figure 2.1

2.2.2 Hardware Architecture

Both the REE and the TEE utilize a number of dedicated resources such as processing core(s), RAM, ROM, cryptographic accelerators, etc. Figure 2-1 provides a simplified example of the resources that may be found at a device

level. Figure 2-2 provides an example of the resources that may be associated with a TEE hosting package such as the System-on-Chip (SoC) in Figure 2-1. Some resources accessible by the REE may also be accessible by the TEE whereas the opposite must not hold unless explicitly authorized by the TEE. The trusted resources are only accessible by other trusted resources and thereby make up a closed system that is protected from the REE.
In general terms, the TEE offers an execution space that provides a higher level of security than a Rich OS; although the TEE is not as secure as an SE, the security it offers is sufficient for most applications.

2.2.3 TEE High Level Security Requirements

The high level security requirements of a TEE can be stated as follows:

- The primary purpose of a TEE is to protect its assets from the REE and other environments.
  - This is achieved through hardware mechanisms that those other environments cannot control.
- This protection must include protection against other execution environments whose software or location is deemed untrustworthy.
- The TEE must be protected against a range of physical attacks, see Protection Profile [TEE PP].
  - Typically this protection will be a level less than that provided to dedicated tamper resistant technology such as Secure Elements.
  - Intrusive attacks that physically break the IC package boundary are outside of the scope of TEE protection.
- The debug facilities of a production TEE (or system components capable of accessing assets in the TEE) must be disabled or must be controlled by an element that itself meets or exceeds the security requirements of the TEE.
  - This requirement places no restrictions on debug capabilities for system components (including the REE) that cannot access assets of the TEE.
- The TEE must be instantiated through a secure boot process using assets bound to the SoC or the Off-SoC Security Processor and isolated from the REE.
  - The integrity and authenticity gained through secure boot:
    - Must extend throughout the lifetime of the TEE.
    - Must be retained through any state transitions in the system such as power transitions or core migration.
- Trusted Storage of data and keys is provided by the TEE.
  - The Trusted Storage must be bound to a particular TEE on a particular device, such that no unauthorized internal or external attacker may access, copy, or modify the data contained.
    - The strength of this protection must be at least equal to that of the TEE environment.

- The Trusted Storage must provide a minimum level of protection against rollback attacks.
  - It is accepted that the actual physical storage may be in an unsecure area and so is vulnerable to actions from outside of the TEE.

- General software outside the TEE must not be able to call directly to functionality exposed by the TEE Internal APIs or the Trusted Core Framework.
  - The non-TEE software must go through protocols such that the Trusted OS or Trusted Application performs the verification of the acceptability of the TEE operation that the REE software has requested.

Detailed definition of the security requirements for a TEE can be found in [TEE PP].

2.3 TEE Resources

A TEE uses three classes of resources.

In-package resource

This type of resource must be implemented in-package, which means that they will be protected from a range of physical attacks. In-package communication channels between these resources do not need to be encrypted as they are considered physically secure.

Off-package, cryptographically protected resource

An exception to the in-package resource can be trusted replay-protected external non-volatile, and trusted volatile memory areas. For these memory areas, the security must be fulfilled by using proven cryptographic methods. Only the TEE will be able to decrypt the plaintext content stored in these locations.

Exposed or partially exposed resources

A further exception is that TEE controlled trusted areas of external components may contain data not guarded by a proven cryptographic method. This is needed to:

- Enable trusted DRAM-based buffers where the data is in the clear but is protected from attack by unauthorized software while being manipulated (e.g., TLS or DRM stream buffers).

- Provide space for a trusted screen frame store.

Neither of the above use cases necessarily requires encrypted RAM storage, just isolation from the REE and other environments.

- Use keyboards and other I/O that are not accessible to the REE but are not guarded from physical attack.

2.3.1 REE and TEE Resource Sharing

The following discussion is simplified to only consider the presence of one TEE and the REE. A TEE is similarly isolated in component ownership and resource sharing from other environments such as SEs and other TEEs.
The REE has access to the untrusted resources, which may be implemented on-chip or off-chip in other components on the PCB. The REE cannot access the trusted resources. This access control must be enforced and can potentially be implemented by physical isolation of trusted resources from untrusted resources. The only way for the REE to get access to trusted resources is via any API entry points or services exposed by the TEE and accessed through, for example, the TEE Client API. This does not preclude the capability of the REE passing buffers to the TEE in a controlled and protected manner and vice versa.

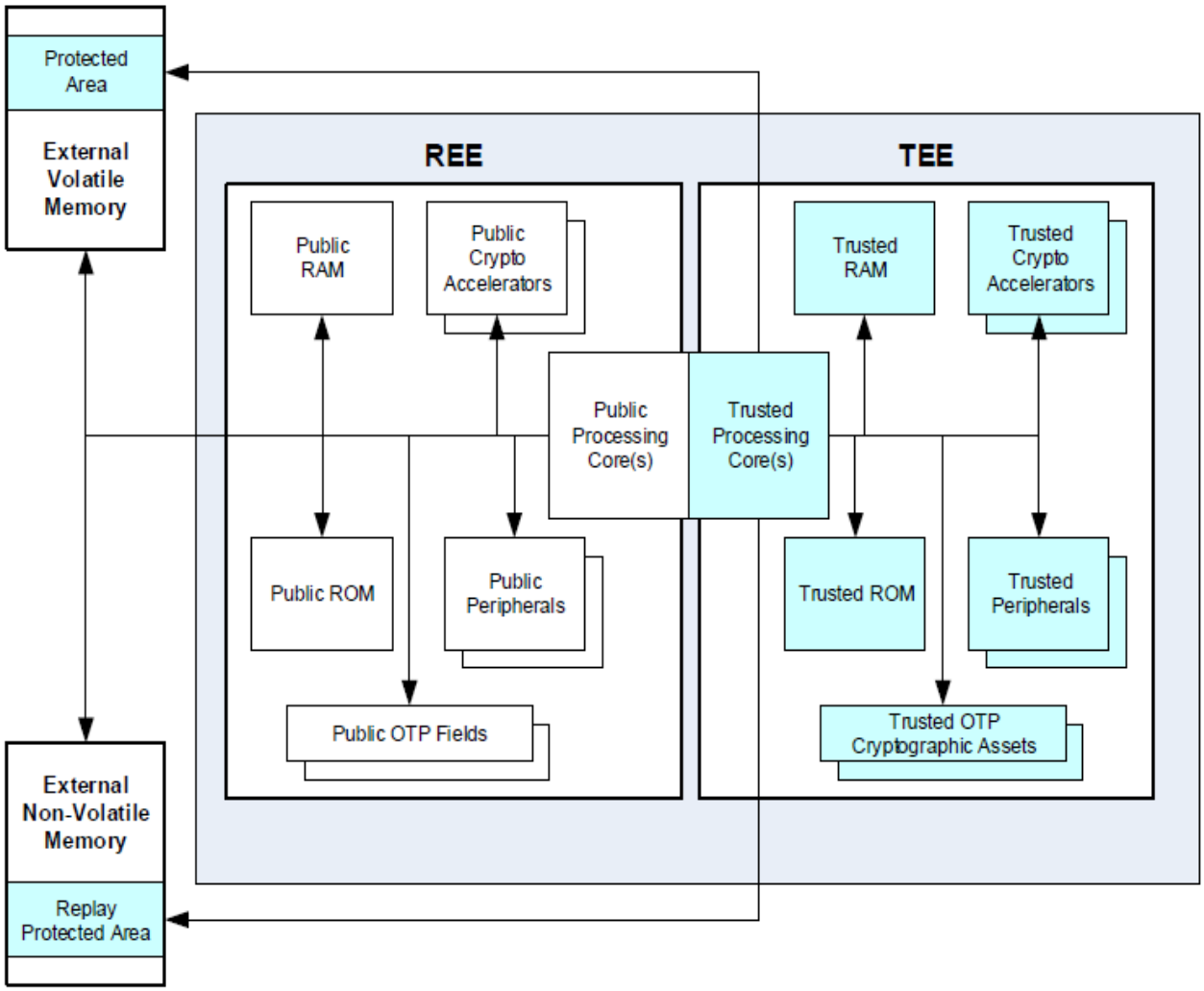


Figure 2.2

Hardware Architectural View of REE and TEE

Note that the architectural view of TEE and REE as illustrated in Figure 2.2 does not dictate any specific physical implementation. Possible implementations include and are not limited to those illustrated in Figure 2.3. Some capabilities may not be supportable by all implementations. For example, PCB A in Figure 2.3 cannot support the Trusted User Interface.

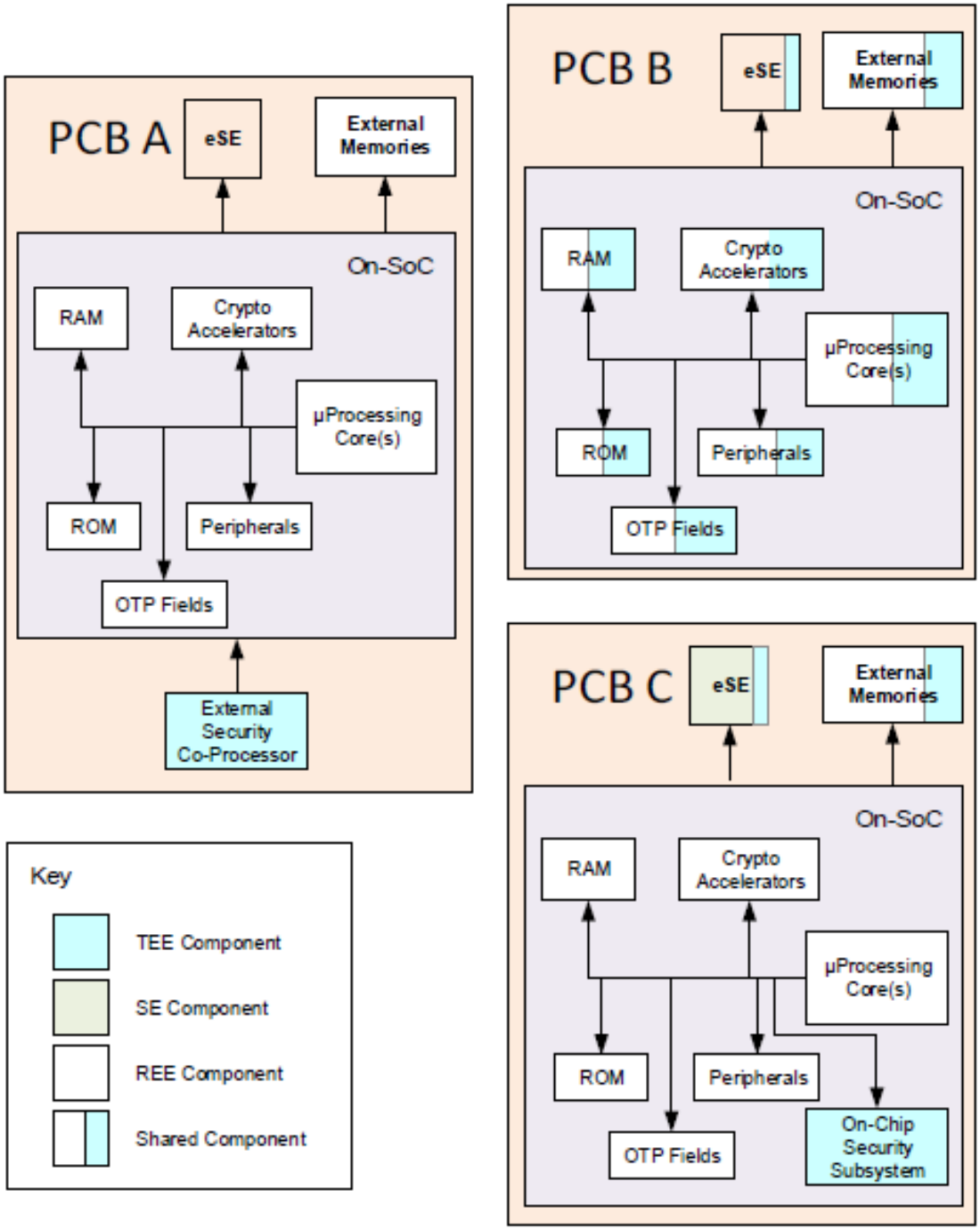


Figure 2.3

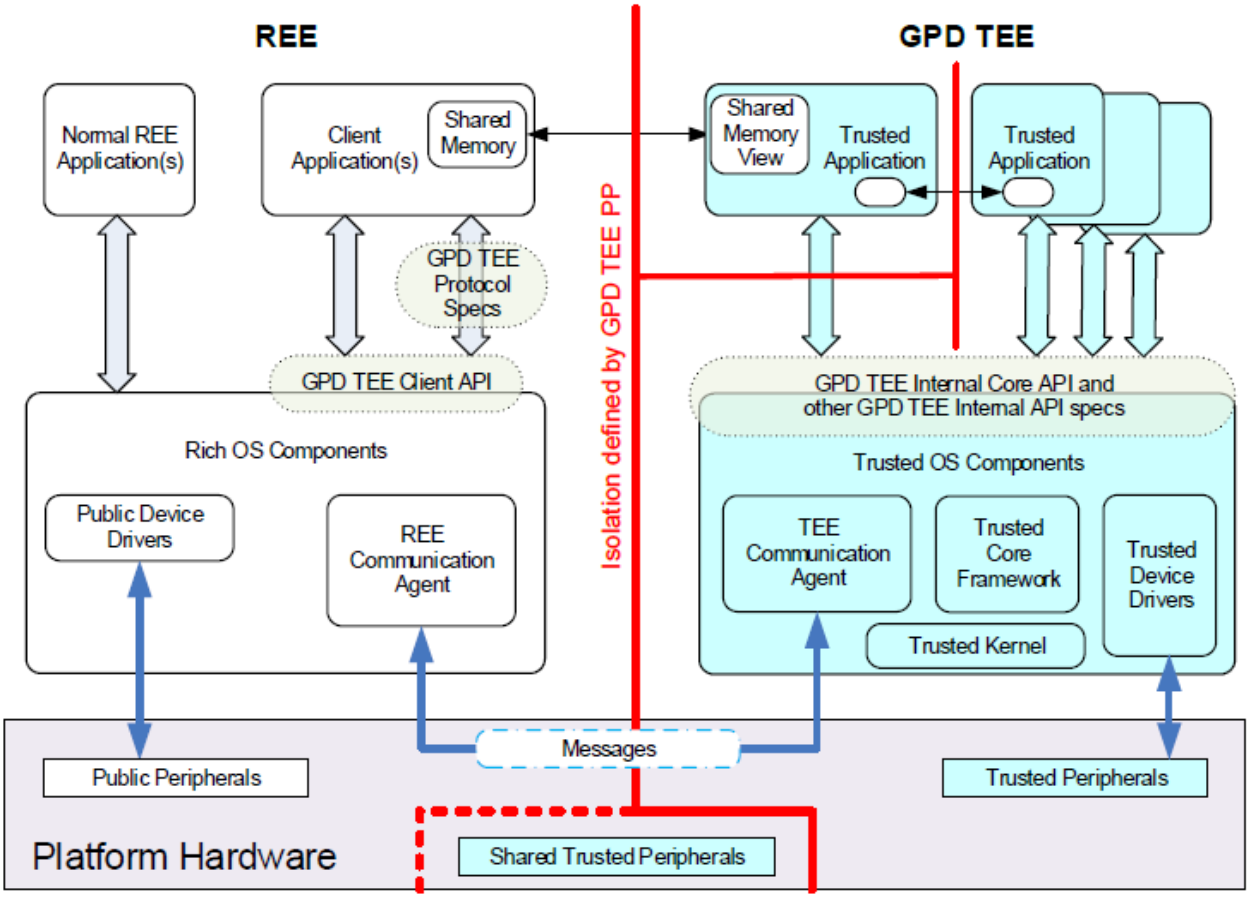
2.4 INTERFACES OF TEE

2.4.1 TEE Software Interfaces

The TEE is a separate execution environment that runs alongside the REE and other environments and provides security services to those other environments and applications running inside those environments. The TEE exposes sets of APIs to enable communication from the REE and others to enable Trusted Application software functionality within the TEE.
This chapter describes the general software architecture associated with the TEE, the interfaces defined by GlobalPlatform, and the relationship between the critical components found in the software system.
There is no mandated implementation architecture for these components and they are used here only as logical constructions within this document.

2.4.2 The TEE Software Architecture

The relationship between the major software systems components is outlined in Figure 2.4



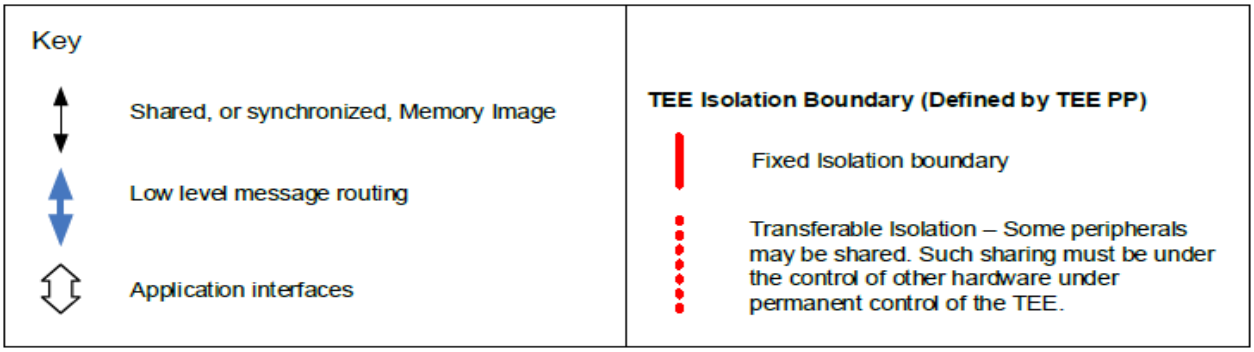


Figure 2.4

TEE Software Architecture

The goal of the TEE Software Architecture is to enable Trusted Applications (TA) to provide isolated and trustworthy capabilities, which can then be used through intermediary Client Applications (CA).

Please note:

- Just as there are many hardware solutions to implementing a TEE (see Figure 2.3: Example Hardware Realizations of TEE) there can also be many software configurations of a TEE (or even TEEs) in a device. The following sections discuss some possible configurations.
- For simplicity, subsequent graphics show only the fixed isolation boundary discussed in Figure 3.1. However, shared trusted peripherals (as illustrated and described in Figure 3.1) are possible in all configurations.

2.5 Components of a GPD TEE

2.5.1 REE Interfaces to the TEE

Within the REE, the architecture identifies an optional protocol specification layer, an API, and a supporting communication agent.

- The REE Communication Agent provides REE support for messaging between the Client Application and the Trusted Application.
- The TEE Client API is a low level communication interface designed to enable a Client Application running in the Rich OS to access and exchange data with a Trusted Application running inside a Trusted Execution Environment.
- The TEE Protocol Specifications in the REE layer offers Client Applications a set of higher level APIs to access some TEE services. TEE TA Debug API [TEE TA Debug] and TEE Management Framework Specification [TEE Mgmt] currently use this stack layer. Additional proprietary TEE APIs

may be developed at the TEE Protocol Specifications layer by TA developers.

2.5.2 Trusted OS Components

Within the TEE, the architecture identifies two distinct classes of software: the hosting code provided by the Trusted OS Components, and the Trusted Applications, which run on top of that code.
Trusted OS Components consist of:

- The Trusted Core Framework which provides OS functionality to Trusted Applications.
  - The Trusted Core Framework is part of the TEE Internal Core API, discussed in section 3.5.
- The Trusted Device Drivers which provide a communications interface to trusted peripherals which are dedicated to the TEE.

Both the Trusted Applications and Trusted Core Framework make use of scheduling and other OS management functions provided by the Trusted Kernel. The Trusted Device Drivers may be an integral part of the Trusted Kernel or may be modular components, depending on the architecture of the Trusted Kernel.

- The TEE Communication Agent is a special case of a Trusted OS component. It works with its peer, the REE Communication Agent, to safely transfer messages between CA and TA.

2.5.3 Trusted Applications (TAs)

The Trusted Applications interface to the rest of the system via APIs exposed by Trusted OS components.

- The TEE Internal APIs define the fundamental software capabilities of a TEE.
- Other non-GlobalPlatform internal APIs may be defined to support interfacing to further proprietary functionality.

When a Client Application creates a session with a Trusted Application, it connects to an instance of that Trusted Application. A Trusted Application instance has physical memory address space which is separated from the physical memory address space of all other Trusted Application instances.
A session is used to logically connect multiple commands invoked in a Trusted Application. Each session has its own state, which typically contains the session context and the context(s) of the Task(s) executing the session.
It is up to the Trusted Application to define the combinations of commands and their parameters that are valid to execute.
TAs may only start execution in response to an external command. They make their own choice as to when to return from that command. Typical TAs follow a short command response life cycle but complex TAs may iterate for long periods while processing input and output events such as TUI.

2.5.4 Shared Memory

One feature of a TEE is its ability to enable the CA and TA to communicate large amounts of data quickly and efficiently via access to a memory area accessible to both the TEE and REE. The API design allows this feature to be implemented by the Trusted OS as either memory copies, or by directly shared memory. The protocols for how to make use of this ability are defined by the TA designer, and enabled by the TEE Client API and TEE Internal Core API.
Care should be taken with the security aspects of using shared memory, as there is a potential for a Client Application or Trusted Application to modify the memory contents asynchronously with the other parties acting on that memory.

2.5.5 TA to TA Communication

A TA may also communicate to another TA. This uses the same process as used by the CA to communicate to the TA, however there is a trustworthy indicator that allows the receiving TA to be assured that communication has not been exposed outside the TEE. This not only simplifies the trust in the communication content but also in the metadata associated with the content, for example providing the identity of the calling TA. Because of this trust relationship, a TA in another TEE in the same device should be treated as though it is an REE based CA, because the receiving TA's TEE has no reason to trust the calling TA's TEE.

2.5.6 Relationship between TEE APIs

To assist the reader in understanding the relationships between the various APIs and released specification documents, the following graphic is provided.

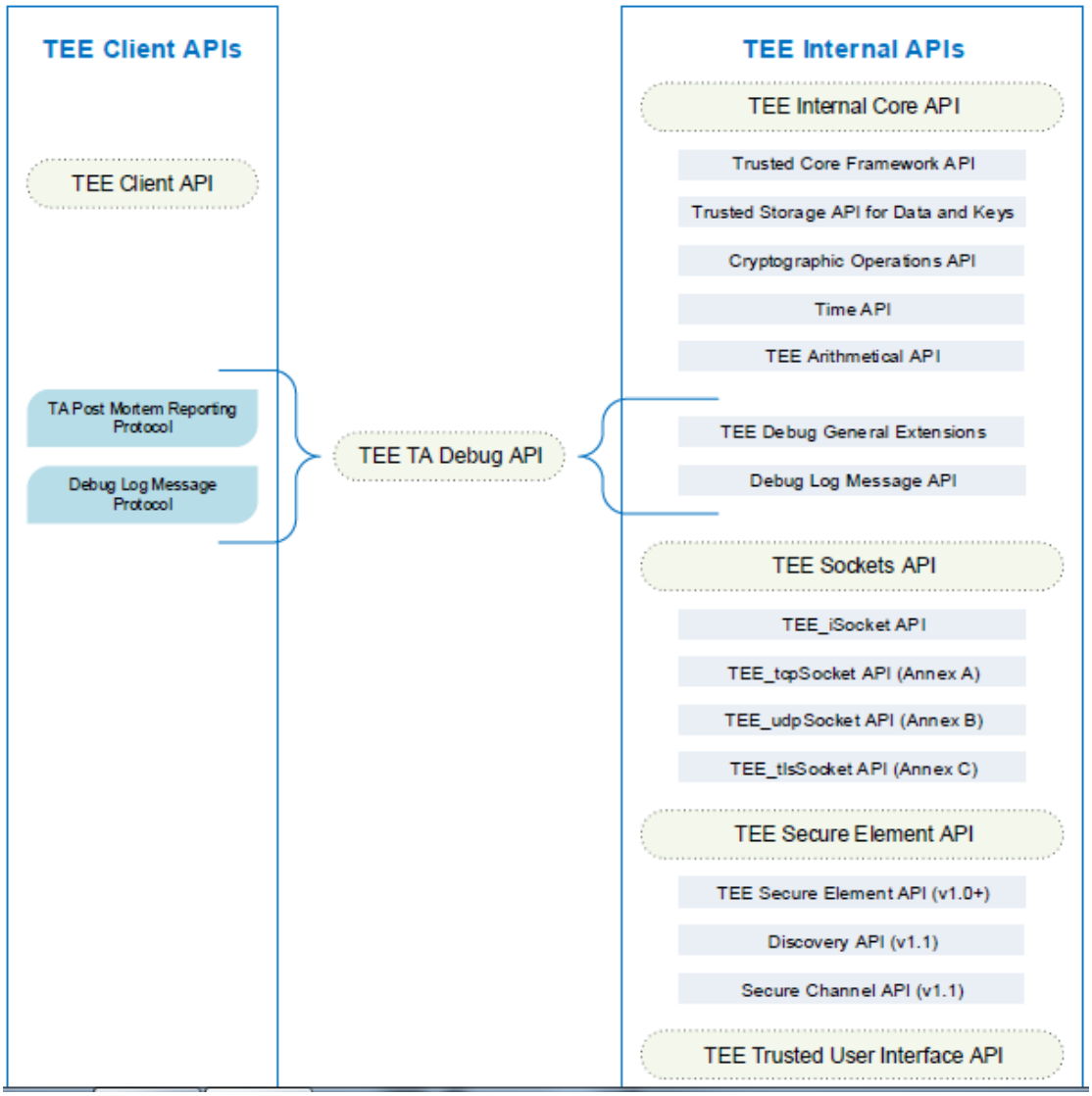


Figure 2.5

TEE APIs

2.6 The TEE Client API Architecture

GlobalPlatform specifies the TEE Client API in the GlobalPlatform TEE Client API Specification [TEE Client API]. The TEE Client API concentrates on the interface to enable efficient communications between a Client Application and a Trusted Application.

Higher level standards and protocol layers (known as TEE Protocol Specifications and functional APIs) may be built on top of the foundation provided by the TEE Client API – for example, to cover common tasks such as trusted storage, cryptography, and run-time installation of new Trusted Applications.

Within the REE this architecture identifies three distinct classes of component:

- The Client Applications, which make use of the TEE Client API

- The TEE Client API library implementation
- The REE Communication Agent, which is shared amongst all Client Applications, and whose role is to handle the communications between the REE and the TEE

2.6.1 The TEE Internal API Architecture

GlobalPlatform specifies a series of APIs to provide a common implementation for functionality typically required by many Trusted Applications. The TEE Internal Core API is specified in the GlobalPlatform TEE Internal Core API Specification [TEE Core API]. The TEE Internal Core API concentrates on the various interfaces to enable a Trusted Application to make best use of the standard TEE capabilities. Additional low level functionality is provided by TEE Internal APIs such as the TEE Secure Element API, TEE Sockets API, and TEE TA Debug API.
Higher level standards and protocol layers may be built on top of the foundation provided by the TEE Internal APIs – for example, to cover common tasks such as creating a trusted password entry screen for the user, confidential data management, financial services, and Digital Rights Management.
Within the TEE, this architecture currently identifies three distinct classes of component:

- The Trusted Applications, which make use of TEE Internal APIs
- The TEE Internal API library implementations
- The Trusted OS Components, which are shared amongst all Trusted Applications, and whose role is to provide the system level functionality required by the Trusted Applications

2.6.2 The TEE Internal Core API

The TEE Internal Core API provides a number of different subsets of functionality to the Trusted Application.

API Name	Description
Trusted Core Framework API	This API provides integration, scheduling, communication, memory management, and system information retrieval interfaces.
Trusted Storage API for Data and Keys	This API provides Trusted Storage for keys and general data
Cryptographic Operations API	This API provides cryptographic capabilities
Time API	This API provides support for various time-based funcionality to support tasks such as token expiry and

	authentication attemp throtting
TEE Arithmetical API	This API provides arithmetical primitives to create cryptographic functions not found in the Cryptographic API

Table 2.1

APIs within TEE Internal Core API

2.6.3 The TEE Sockets API

The TEE Sockets API provides a common modular interface for the TA to communicate to other network nodes, acting as a network client. The TEE Sockets API is the general API for accessing and handling client sockets of various kinds.

- TEE Sockets API Annex A specifies the TEE_iSocket interface for Transmission Control Protocol (TCP).
- TEE Sockets API Annex B specifies the TEE_iSocket interface for User Datagram Protocol (UDP).
- TEE Sockets API Annex C specifies the TEE_iSocket interface for Transport Layer Security (TLS).

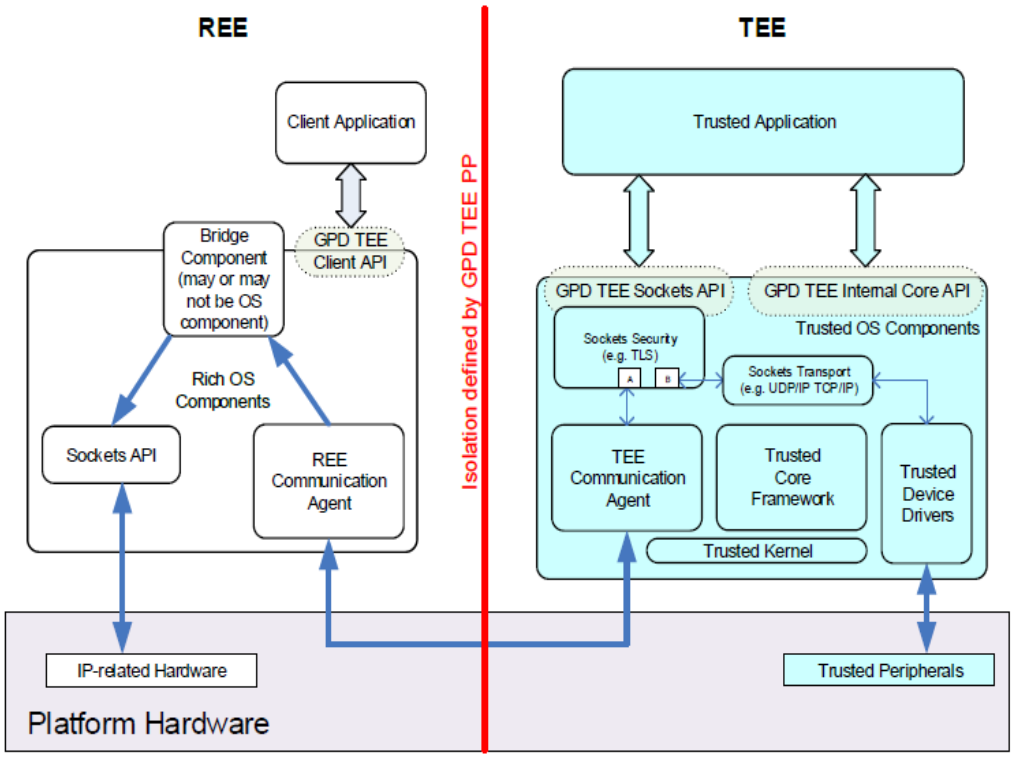


Figure 2.6

The above diagram shows two routing options (A) and (B) inside the TEE. These options are shown because only the security layer MUST reside inside the TEE. In a real implementation it would be expected that only one of these options (A) or (B) would be needed and typically functionality such as UDP/IP and TCP/IP brings no security risks by placing in the REE.
More information on the TEE Sockets API can be found in the GlobalPlatform TEE Sockets API Specification [TEE Sockets].

2.6.4 The TEE TA Debug API Architecture

The TEE TA Debug API provides services that are designed to support TA development and/or compliance testing of the TEE Internal APIs.
The Post Mortem Reporting (PMR) service supports compliance testing and TA debug. This service provides a method for a TEE to report to clients the termination status of TAs which enter the Panic state. Without this capability it is not possible to certify correct functionality of the TEE internal APIs, as the Panic state is used to report various error conditions that need to be tested.
The Debug Log Message (DLM) service is useful in a TA debug scenario. This service provides a method for a TA to report simple debug information on authorized systems. It may report to client applications or off-device hardware or both. Compliant implementations should implement the DLM interface but it is not mandatory to do so.
More information of the TEE TA Debug API Architecture can be found in the GlobalPlatform TEE TA Debug Specification [TEE TA Debug].

2.6.5 The TEE Secure Element API Architecture

The TEE Secure Element API is an enabling thin layer to support communication to Secure Elements (SEs) connected to the device within which the TEE is implemented. This API defines a transport interface based on the SIMalliance Open Mobile API specification [Open Mobile].
SEs may be connected to the REE or exclusively to the TEE.

- An SE connected exclusively to the TEE is accessible by a TA without using any resources from the REE. Thus the communication is considered trusted.
- An SE connected to the REE is accessible by a TA using resources lying in the REE. It is recommended to use a secure channel (i.e. by using the Secure Channel API) to protect the communication between the TA and the SE against attacks in the REE.

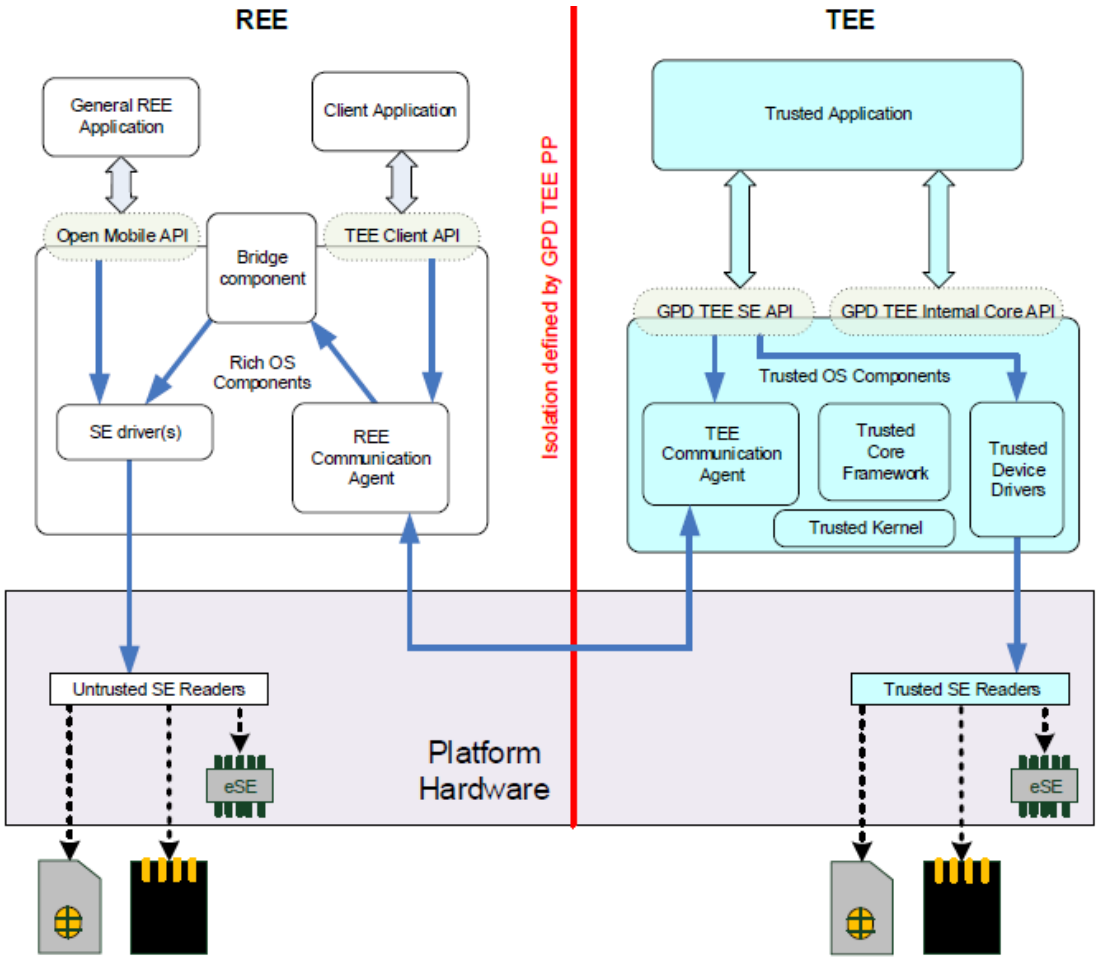


Figure 2.7
Typical Device with Multiple SE Readers

More information about the TEE Secure Element API can be found in the GlobalPlatform TEE Secure Element API Specification [TEE SE API].

2.7 The TEE Trusted User Interface API Architecture

The Trusted User Interface API permits the display of screens to the user and achieves three objectives:

- Secure display – Information displayed to the user cannot be accessed, modified, or obscured by any software within the REE or by an unauthorized application in the TEE.
- Secure input – Information entered by the user cannot be derived or modified by any software within the REE or by an unauthorized application in the TEE.
- Security indicator – The user can be confident that the screen displayed is actually a screen displayed by a TA.

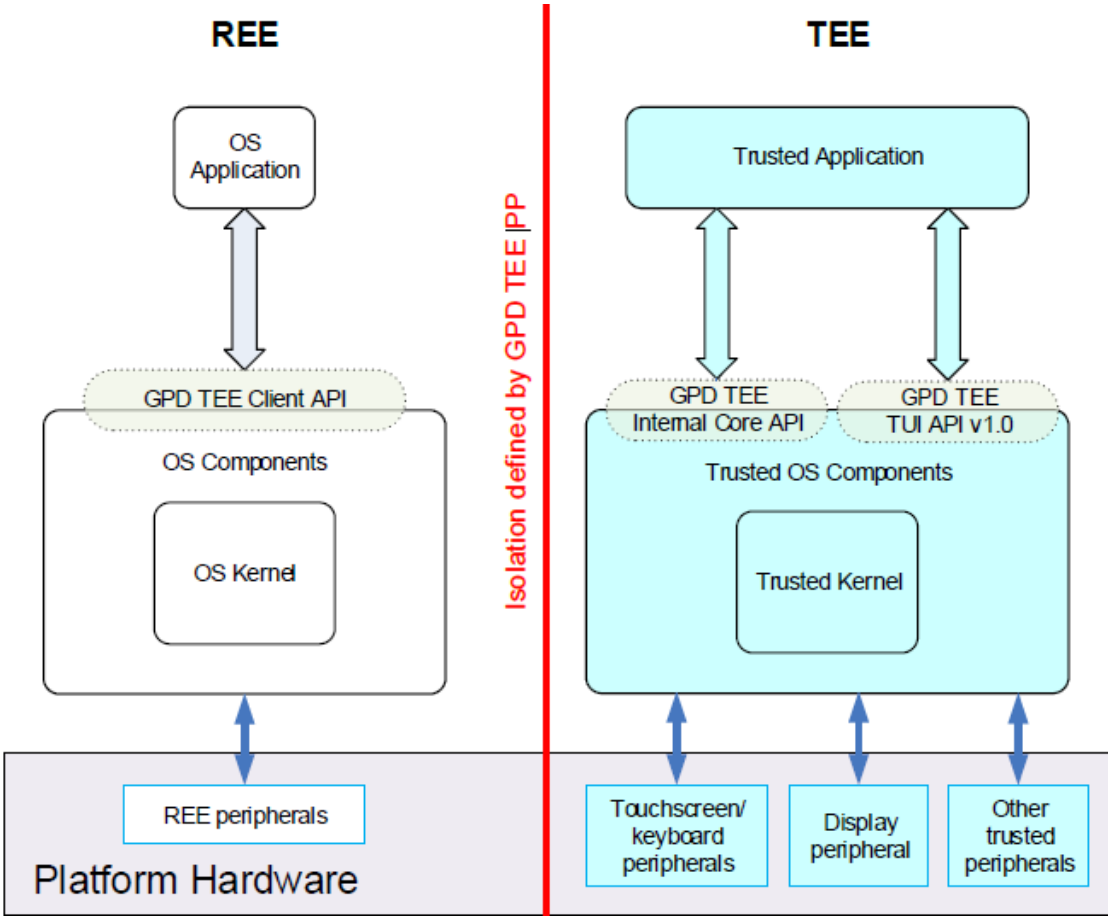


Figure 2.8
TEE with TUI Architecture

While it cannot be assured that the user will not enter his identifying secret into an REE UI session, it can be assured that such an REE UI session will not have access to the secrets of the TEE. As such it is not the user's identification that should be treated by remote parties as a trustworthy identification, but that identification in combination with a factor only known to the TA in the TEE (such as a key).
Another assurance that a third party gains from using the TEE TUI is the guarantees of non-interference. This allows remote party confidence that what the user signs is what they actually saw, and not some information spoofed into the UI, replacing the desired display information.
More information about the TEE Trusted User Interface can be found in the GlobalPlatform TEE Trusted User Interface API Specification [TEE TUI API].

2.8 Variations of TEE Architecture Found on Real Devices

Real devices will contain extensions to the basic TEE architecture, and may potentially house multiple TEEs. The GlobalPlatform TEE Protection Profile [TEE PP] requires that a TEE, including its proprietary extensions, is isolated from other environments including other TEEs.

2.8.1 A GlobalPlatform Compliant TEE May Have Proprietary Extensions

A compliant GPD TEE may offer additional APIs to Trusted Applications, and may offer other access methods to REE applications. This allows flexibility in implementation in special markets, and provides a route for growth of the GlobalPlatform TEE specifications as new APIs are found to be useful and hence taken up by GlobalPlatform as new TEE specifications.

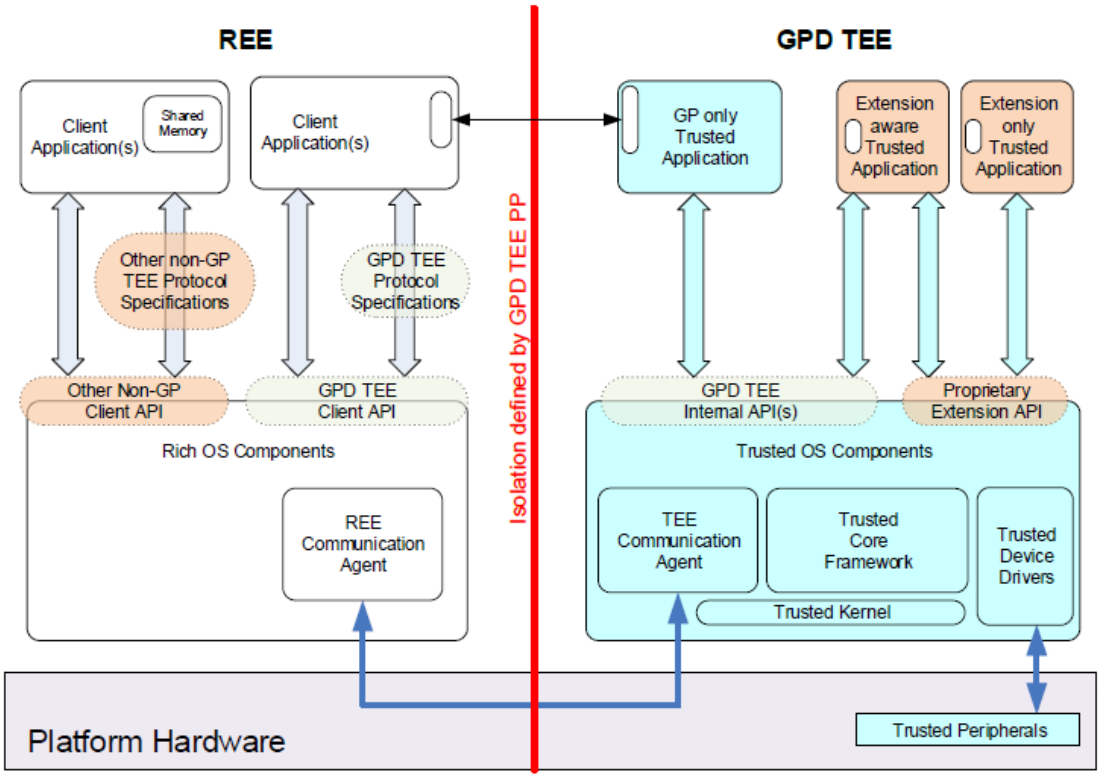


Figure 2.9

Please note:

- This is one example configuration of a proprietary extension of a compliant GPD TEE, and other configurations may exist.
- There is no specified limitation on the number of OSes in an REE or the number of TEEs in one device.
- Shared trusted peripherals (as illustrated and described in Figure 3.1) are possible in the configuration shown in Figure 3.6.

2.8.2 A Device May Have Many TEEs

There is no specified limitation on the number of TEEs in a device. The TEE Client API provides a methodology for an REE application to communicate to a specified GPD TEE.
For example, a device may have hardware such as shown in Figure 3-7. The illustrated device has three TEEs, each created using a different example method. Each must have an independent set of innately trusted components, and will be isolated from the other TEEs and the REE, at least at the level of the GlobalPlatform TEE Protection Profile ([TEE PP]).

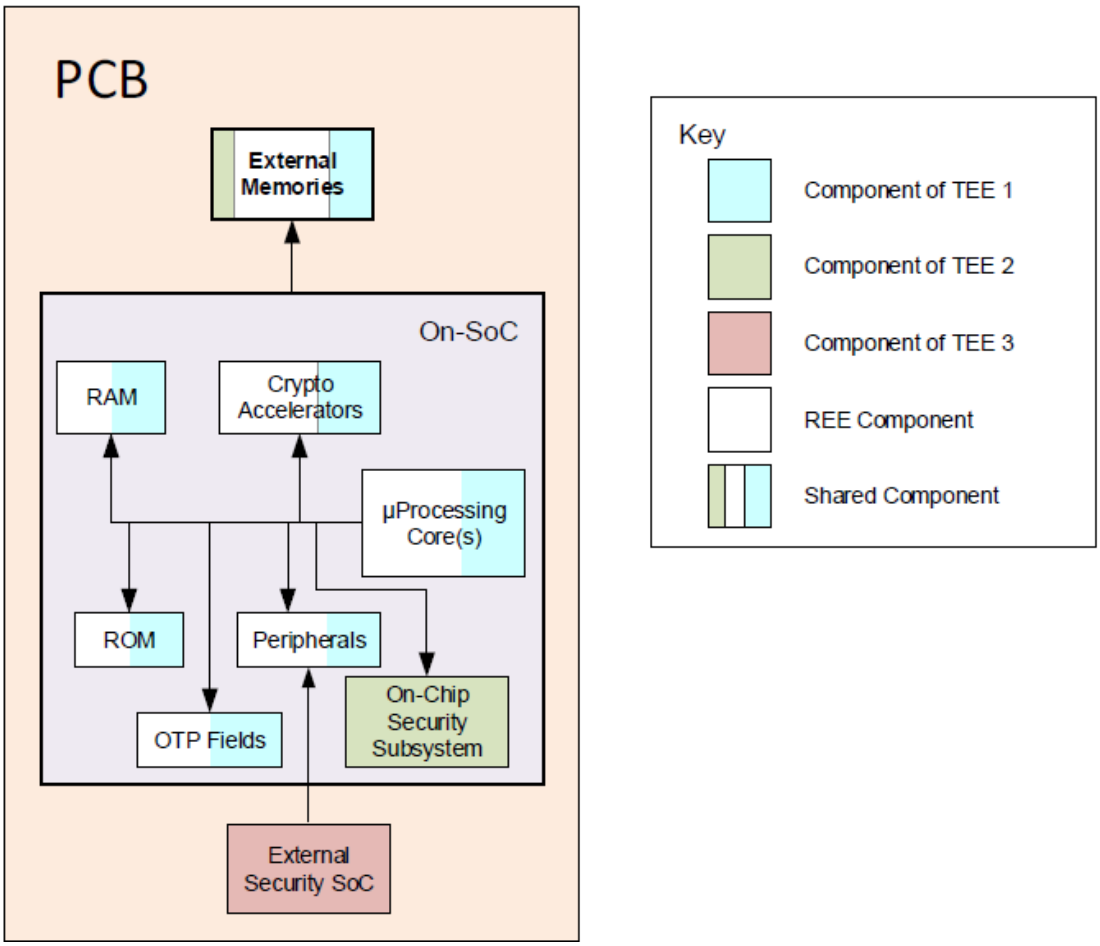


Figure 2.10

Note that inside one GlobalPlatform TEE Protection Profile boundary there can only be one Trusted OS and hence one set of TEE resources.

This does not prevent the GPD TEE sharing resources with other TEEs in much the same way as it may do with the REE. For example, a Trusted User Interface is typically owned in an untrusted mode by the REE and only taken over by the TEE and put in a trusted state when needed. With multiple TEEs, such a Trusted UI would potentially be shared between all TEEs and the REE with only one having active ownership at one time.

Communications between TEEs must be treated by a Trusted Application on the initial supposition that the endpoint is untrusted (in the same way as the TA should treat anything outside its local TEE).

Figure 3.8 shows an example with two GPD TEEs (i.e. two TEEs that are compliant with a GPD TEE functionality configuration and certified according to the GlobalPlatform TEE Protection Profile). Each GPD TEE exists within its own isolation boundary and does not trust components outside of the boundary. Therefore, from the viewpoint of GPD TEE (a), GPD TEE (b) must be assumed to be untrustworthy as it is not part of GPD TEE (a). Likewise, GPD TEE (b) trusts neither the REE nor GPD TEE (a).
If any OS in the device wishes to use a GPD TEE based set of innately trusted components to secure its boot, then it is up to the boot structure of that Rich OS (i.e. its BIOS, UEFI, etc.) to choose which GPD TEE it uses. Potentially in a system with more than one Rich OS, each may choose to use a different TEE.

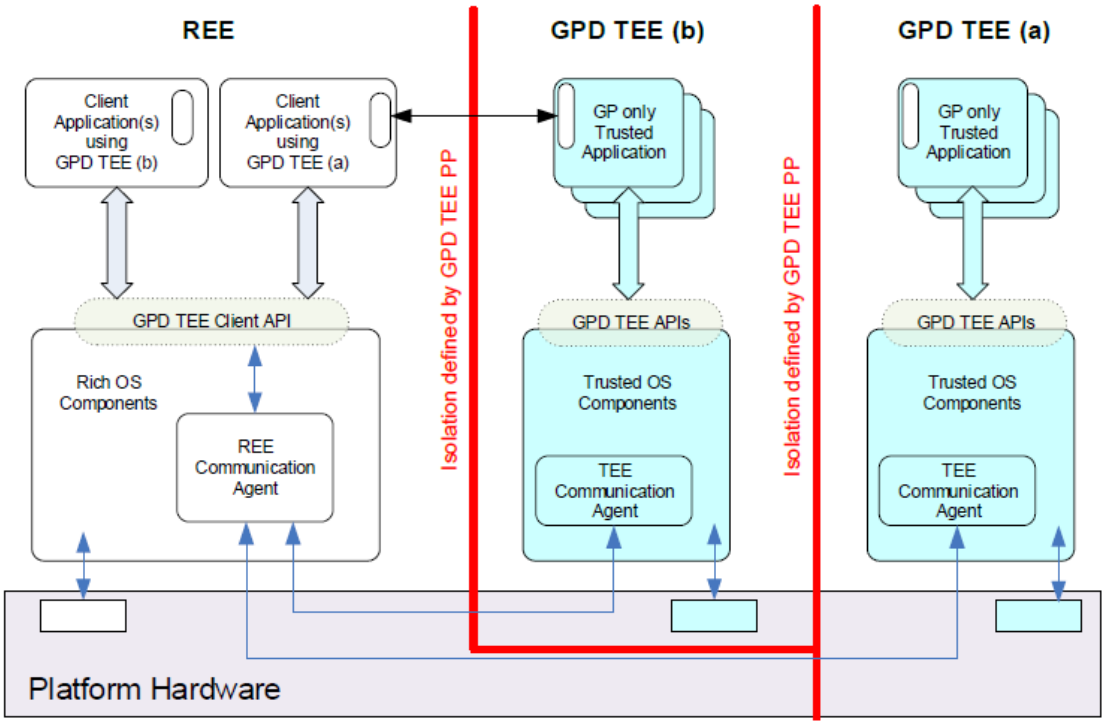


Figure 2.11

Please note:

- This is one example configuration of a system with two GPD TEEs, and other configurations may exist.
- There is no specified limitation on the number of TEEs and OSes in the REE in one device.
- Shared trusted peripherals (as illustrated and described in Figure 3-1) are possible in the configuration shown in Figure 3.8.

2.8.3 Not All TEEs on a Device Need To Be GlobalPlatform Compliant

A device may even have environments that claim to be TEEs but are not GlobalPlatform compliant TEEs.
Clearly if such an environment does not meet GlobalPlatform specifications then GlobalPlatform cannot make any assertions about that environment; however the environment does not raise an issue because a compliant GPD TEE will still be isolated from it as specified in the GlobalPlatform TEE Protection Profile [TEE PP].

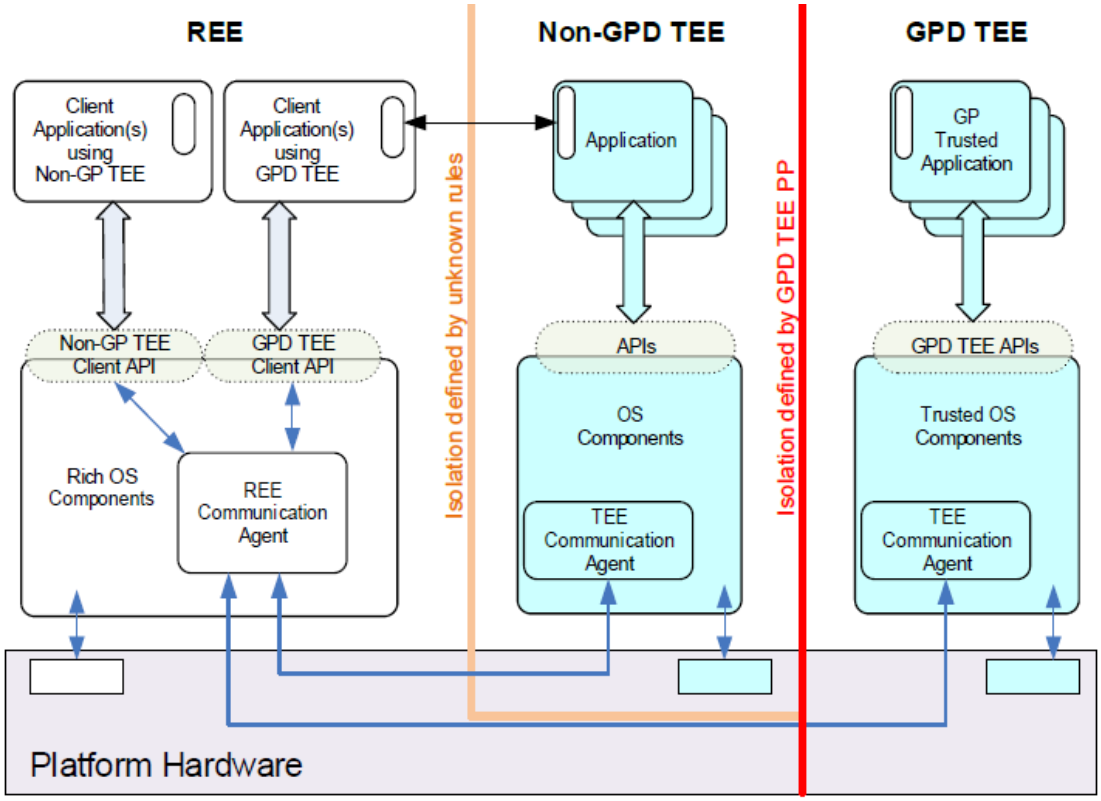


Figure 2.12

GPD TEE alongside Unknown TEE

Please note:

- This is one example configuration of an unknown TEE alongside a GPD TEE, and other configurations may exist.
- There is no specified limitation on the number of GPD TEEs, non-GlobalPlatform TEEs, and OSes in the REE in one device.
- Shared trusted peripherals (as illustrated and described in Figure 3.1) are possible in the configuration shown in Figure 3.9.

3. TEE Management

3.1 TEE Management

Management of the TEE and Trusted Applications running in the TEE is described in the TEE Management Framework specification [TEE Mgmt]. The remote management life cycle of Trusted Applications, GlobalPlatform style management Security Domains, and the TEE itself are detailed in that specification as well.
Each GlobalPlatform style Security Domain (SD) has a nominal off-device "owner" with rights to control those SDs and TAs directly and indirectly below the given SD. The exception to this is when the child SD is a root SD (rSD) because root SDs form a management isolation boundary which limits parental interference.
The TEE Management Framework provides means to securely manage Trusted Applications in a TEE. The following three layers are described.

Administration operations

- Defines the set of supported operations to manage Trusted Applications, the conditions of use, and the detailed behavior of each operation.

Security model

- Defines who the actors are and how the different business relationships and responsibilities can be mapped on the concept of Security Domains with privileges and associations.
- Defines the security mechanisms used to authenticate the entities establishing a communication channel, to secure the communication, and to authorize the administration operations to be performed by Security Domains.
- Defines schemes for key and data provisioning and describes the associated key management.

Protocols

- Defines the command set (over the TEE Client API) to be used to perform the administration operations.
- Defines the command set to be used to establish a secure session with a Security Domain.

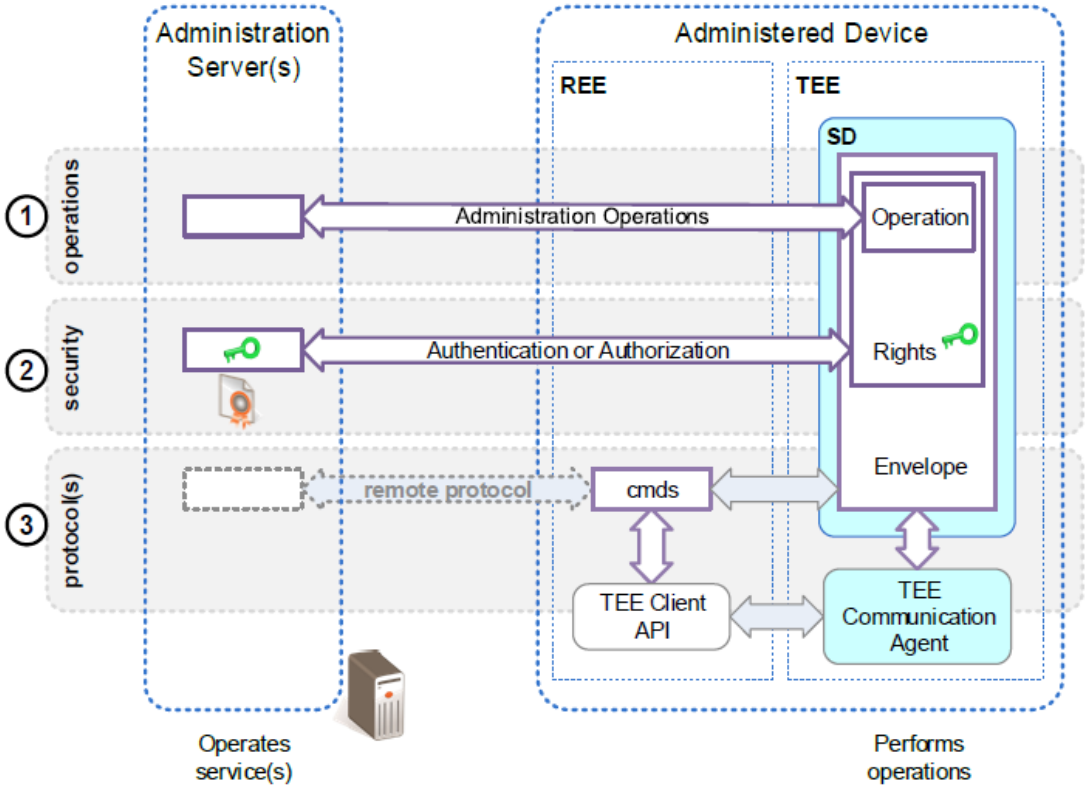


Figure 3.1

TEE Management Framework Structure

The following diagram shows an example of possible management relationships between Security Domains and between Security Domains and Trusted Applications enabled by the TEE Management Framework specification ([TEE Mgmt]).

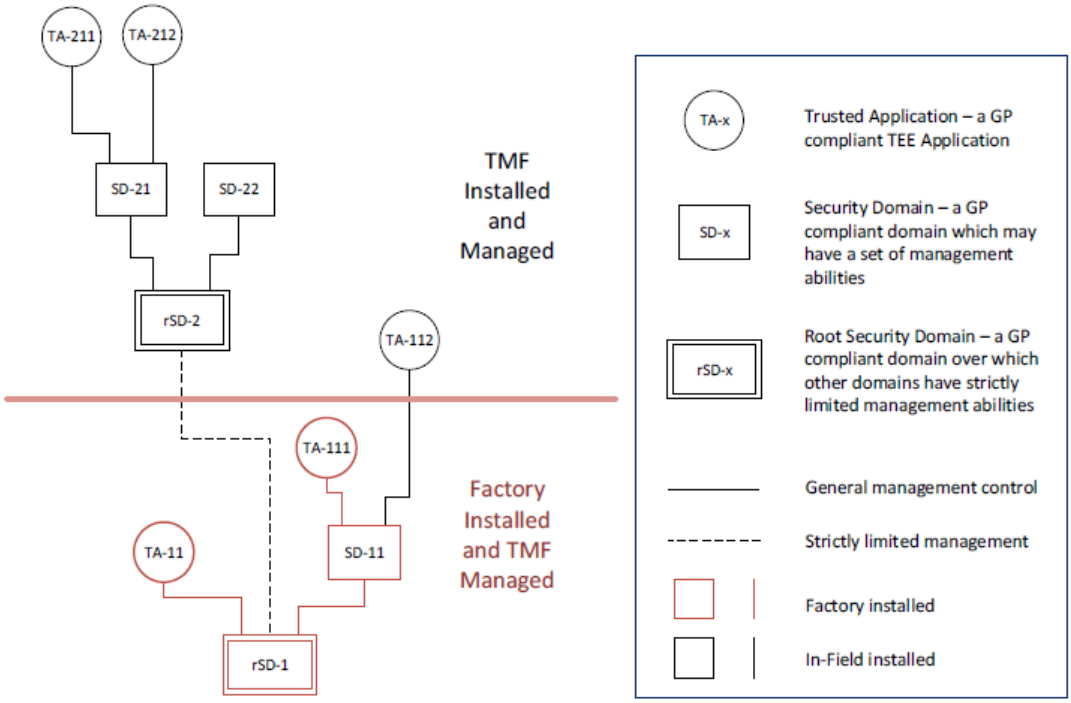


Figure 3.2

Security Domain Management Relationships

The above diagram is just an example of how a management structure may be developed on a platform.
In Figure 4.2:

- rSD-1 is the direct parent of TA-11
- rSD-1 is the indirect parent of TA-111
- The owner of rSD-1 can potentially control any SD-1* or TA-1* but has no control of any of the other current SDs on the example platform, due to rSD-2.
- The owner of rSD-1 can install rSD-2 but may not interact with any of rSD-2's direct or indirect children and is strictly limited in the operations it can perform on rSD-2

There are some exceptions to the above rules, such as with regard to factory reset. For more detail see [TEE Mgmt].

The draft [TEE Mgmt] places no restriction on the number of SDs or TAs that may be installed in the factory, or in the field. Particular platform implementations will have limits on available storage resources and these limits will affect the numbers of TAs and SDs that might be deployed on that platform.

updating, blocking, and personalization. Particular platforms may choose to restrict the availability of certain TEE Management Framework management operations on that platform and similarly particular Security Domains may choose to limit the operations available to their child Security Domains.
Future specifications from GlobalPlatform are expected to provide defined configurations for particular sorts of devices (e.g. IoT and smartphone), enabling those interested in developing and managing TAs to understand the minimum expectations on the sort of TA and SD management structures that might be created on those devices.

3.2 TEE Implementation Considerations

The TEE and its capabilities will be closely coupled to the capabilities of the REE and the state of the device it resides in. It is therefore important for the developer of those REE Client Applications, and even the Rich OS itself, to understand the availability of the TEE capabilities, along with the general security states (and hence vulnerabilities) that may be found in typical devices. Toward that end, this chapter lists some of the possible device states and discusses the notions of Boot Time Environment and Run Time Environment. Some clarifications are given regarding dependencies and the availability of TEE functionalities with respect to the Rich OS.

3.3 Device States

Devices implementing a TEE can be found in a number of states that are not defined in GlobalPlatform specifications, but that are still useful for the developer to understand.
Devices implementing the TEE must provide trusted mechanisms control the corresponding security environments and transitions.
The specific implementations and characteristics of these and other similar states are up to the device manufacturers and the OEMs.
Examples of some such states:

- Devices in manufacturing, which may offer neither security nor functional compliance at various stages of their creation.
- Development devices, which may or may not have reduced security but should provide TEE compliant functionality.
- Production devices, which must provide TEE compliant functionality and security.
- And finally, devices that have somehow failed, and which must block access to TEE held user data, while enabling various levels of debug access through secure mechanisms.

Life cycle state changes must not lower the security of the TEE.

3.4 Boot Time Environment

The term “boot time” refers to the time frame from the reset/power-up of the underlying hardware to the time an operating system has completed its initialization and loading. Based on this definition, boot time software also includes any firmware/ROM code that takes over the control of execution after the device is reset.
The integrity of the initial trusted boot code is intrinsically guaranteed. Furthermore, flexible trusted boot requirements and OEM-dependent boot operations require that, during boot time, some services or operations need to be performed in a trusted execution environment. Hence a minimal set of the TEE capabilities must exist during the device boot time and to enable some of these services a Trusted OS (or some simplified version thereof) may also exist.
A typical TEE secure boot is based on three key components:

- A fixed set of innately trusted components, which typically is the smallest distinguishable set of hardware and/or software that must be inherently trusted and tied to the logic/environment where trusted actions are performed.
- Immutable boot software that is stored, for example, in in-chip TEE ROM
- and the isolated TEE where this security critical boot software is executed.

It is not the current intention of GlobalPlatform to define the boot time capabilities of a TEE, however if a TEE Trusted OS is required to function during boot then it is recommended for compatibility and ease of development that it implements as much of a subset of the TEE Internal APIs as it is capable of providing.

3.4.1 Typical Boot Sequence

Figure 5.1 illustrates three simplified examples of using secure boot of a TEE. Common to all solution examples, the device boots from the TEE boot ROM code inside the SoC containing the TEE (which may not always be the SOC containing the REE). The TEE boot ROM may then load further firmware components and verify them before execution. To verify them, code in the boot ROM will use the information found in the fixed set of innately trusted hardware components (for example, based on information stored in the TEE boot ROM or one-time programmable (OTP) fuses). The firmware components are typically stored in rewriteable non-volatile memories such as flash storage but may also be part of the TEE ROM code.
Before exiting the secure boot process, the firmware or the TEE platform code loads and may verify REE boot loader(s) before their execution. Typically, if any loaded software component verification fails up to this point, the boot process halts and the device reboots with a possible error report/indication. In a successful case, the REE boot loader starts the process of loading the Rich OS or further boot loader components.

OEMs may differentiate by implementing trusted firmware to be run early in the boot sequence. This gives the OEM the flexibility to bring in its own keys, certificate format, signature schemes, etc. Figure 5.1 through Figure 5.3 illustrate example boot sequences, and others may exist.

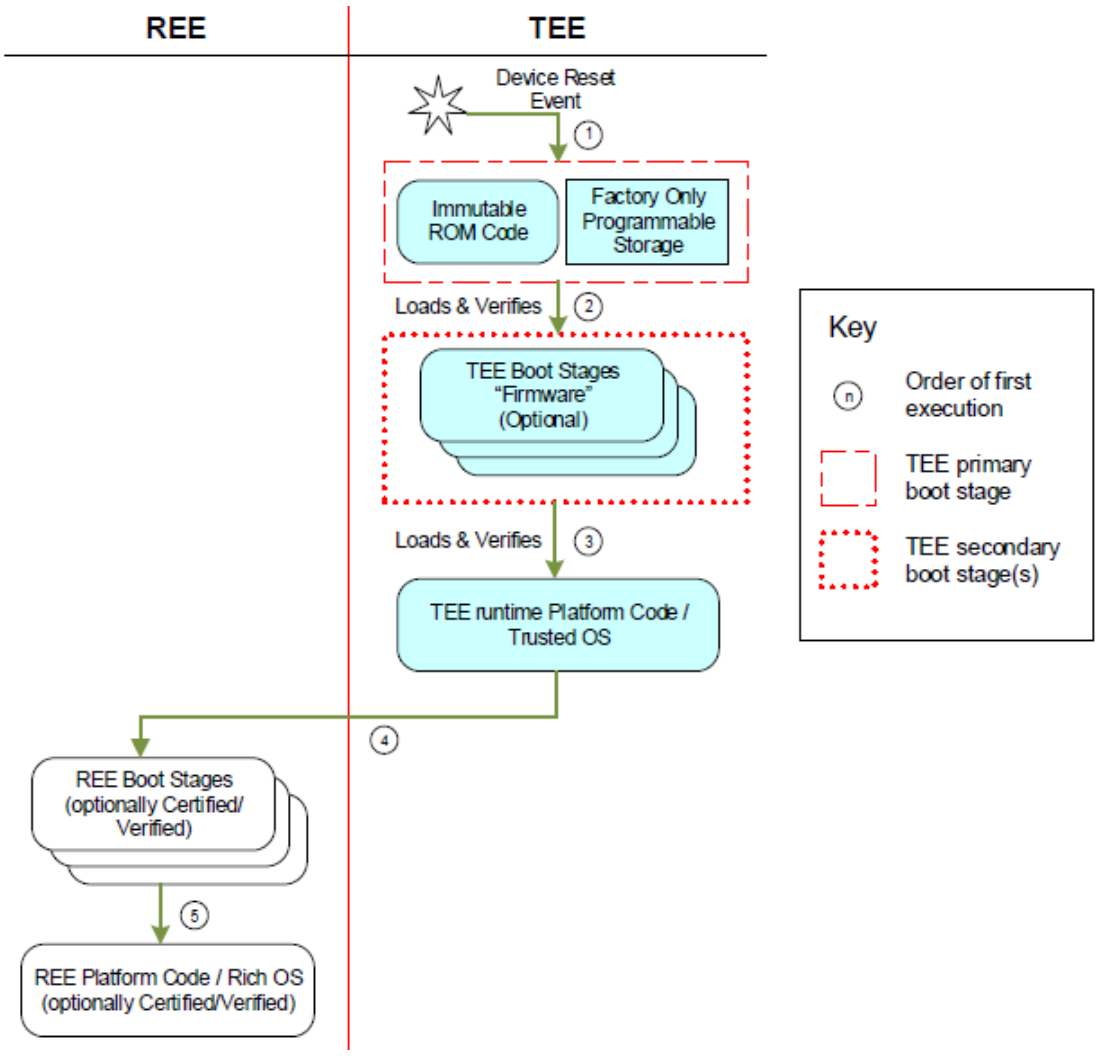


Figure 3.3

Boot Sequence : Trusted OS Early Boot

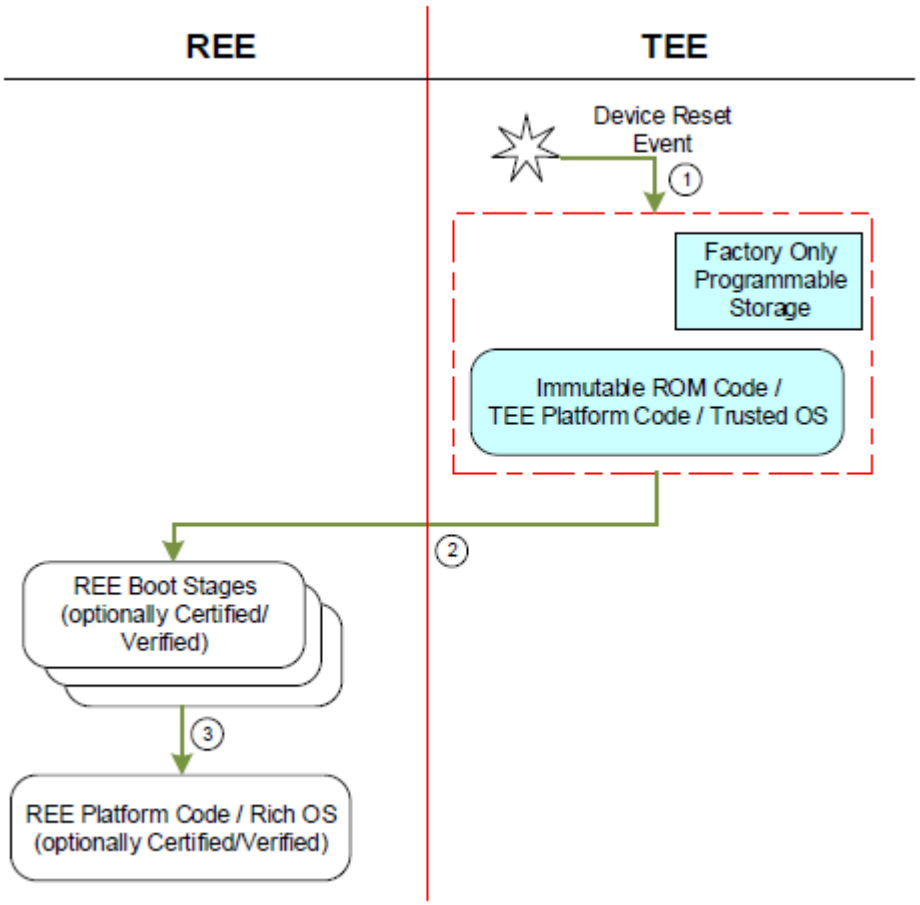


Figure 3.4

Boot Sequence: ROM based Trusted OS

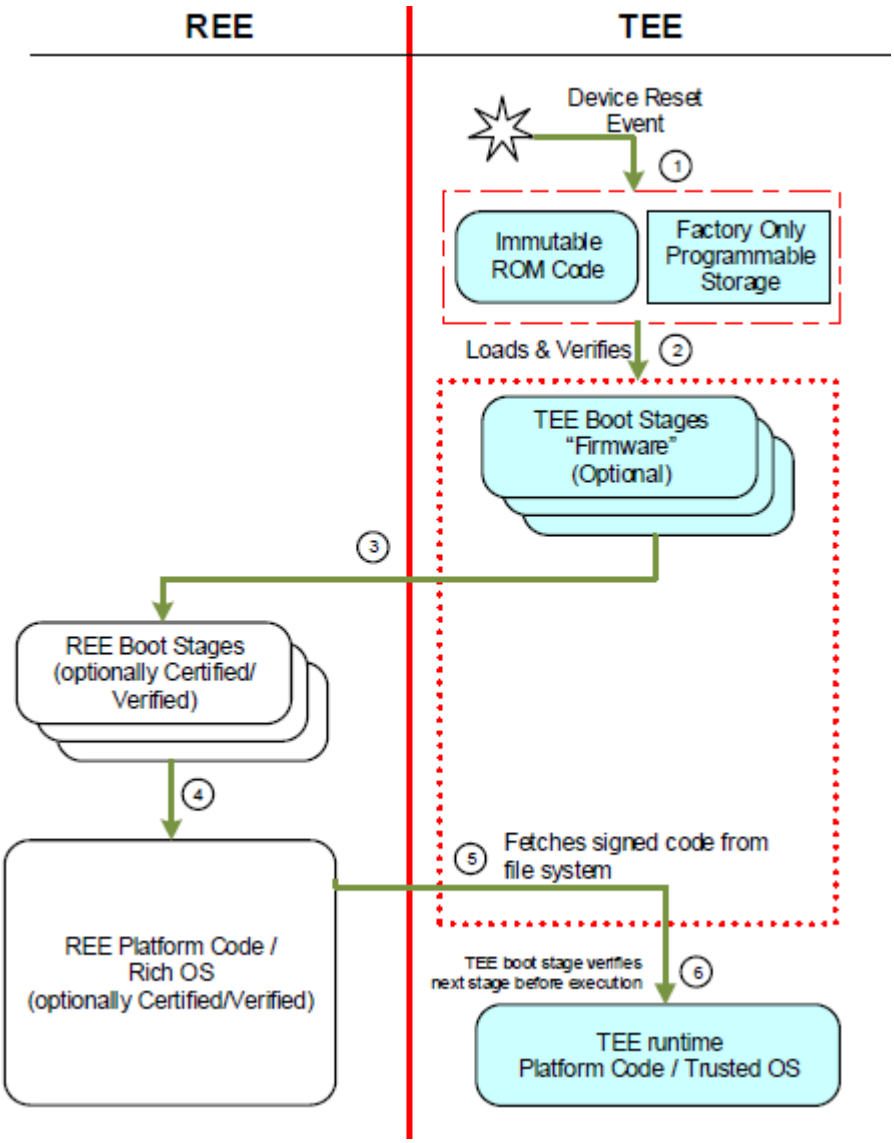


Figure 3.5

Boot Sequence: Trusted OS On-demand Boot

3.5 Run-Time Environment

The term "run-time" refers to a property of the overall execution environment where an operating system has fully completed its initialization/boot operations and is fully operational, as opposed to the duration where the operating system is not fully operational as explained in section 5.2.
The dependencies between the Trusted OS and the Rich OS are implementation dependent. Current GlobalPlatform specifications standardize the behavior of

the system once the Rich OS is operational. This does not mean that there are no capabilities when the Rich OS is not operational, see section 5.2.

3.5.1 TEE Functionality Availability

While the TEE as a protected environment will always meet its protection requirements, its functionality and availability may have dependencies on the REE.
The TEE functionality (i.e. providing GlobalPlatform compliant response to Client API or Internal API commands) is guaranteed to be available whenever the REE is available for REE Client Applications.
The above guarantee of availability to client applications means that effects such as power state changes, where the Client Applications are not aware of such a change, must not be noticeable via their connection to Trusted Applications unless the Trusted Applications chooses to expose such information.

4. OPEN-TEE

4.1 OPEN-TEE

Hardware-assisted TEEs have been available in mobile devices for almost a decade, but access to this technology has been granted only to privileged developers. For example, developers working for chip vendors, OEMs and Original Design Manufacturers (ODMs). Limited access for third-party developers can be accounted for by a variety of reasons, the technology is proprietary, lack of trust of the third party, lack of an easily deployed Software Development Kit (SDK). In addition, there is no unified means to access the resources and functionality of the TEEs.

In order to pave the way for the widespread use of TEE functionality by developers and researchers this thesis defines an architecture and SDK. It is implemented as a framework atop a set of tools that are familiar to the developer, thus removing the need for specialized hardware and the overheads that it incurs.

4.2 Motivation

4.2.1 Enable developer access to TEE functionality

For a variety of reasons, access to TEEs is generally restricted to developers working for chip manufacturers and for the OEMs that make devices based on these chips. Usually, the technology is proprietary and easily deployable SDKs are not available. Furthermore, TEEs may not have a security architecture within them to safely allow complete outsiders access to the them without impairing overall security. However, there have been attempts to address this problem [24].

4.2.2 Provide a fast and efficient prototyping environment

The most common methods of debugging TAs are to either use expensive Joint Test Action Group (JTAG1) debugging or resort to primitive "print tracing" by inserting diagnostic output in the source code. The former generally allows for detailed instruction level debugging. However, the costs associated with these debuggers can be prohibitively expensive, and the setup complex. Print tracing as a debugging technique is cumbersome and clutters up the source code even to locate the source of a problem. Another concern encountered by TEE developers is that if a TA running on actual device hardware crashes, a hard reset of the device maybe required to recover, thereby significantly increasing the time and effort of debugging.

4.2.3 Promote research into TEE services

Ways to isolate TEEs from REEs are reasonably well understood as we saw in

Section 2. What is less well understood are the types of services that could benefit from using TEEs. As the app store model2 has proven, given an opportunity, the developer community at large is capable of pushing the boundaries and exploring new and novel ways to use technology. Making it possible for researchers to easily develop TAs could trigger the development of novel and innovative applications.

4.2.4 Promote community involvement

The prerequisite for involving the developer community and researchers at large is to allow them access to a freely and easily available development environment, SDK and a platform with which to experiment. The financial and technical aspects of making hardware TEEs available for development on a large scale motivates the need for a software framework for TA development which is not bound to any particular hardware or vendor.

To get the community involved, application developers need development environments, SDKs and a platform with which to experiment. Although the GP standard simplifies conceptualizing a TEE and the functionality that can be offered by it, there are a number of obstacles that the developer must first overcome. The hardware is complex and expensive to design and manufacture, by its nature it is complicated to deploy, test with and there are no standard tools with which to work.

Safely exposing TEE functionality to application developers will enable them to innovate these novel approaches to improve the security and privacy of their applications. Exposing TEE technology to a wider audience in no way guarantees that all security threats will disappear, however, a community can provide more varied research and ideas than a small group ever could. This is one of the few ways that we can start to realize the potential of a TEE. The financial and technical aspects of accessing a hardware TEE make this infeasible so we propose the creation of a TEE framework, that is not bound to any hardware or any particular vendor, yet tries to conform to one standardization effort, for which we have chosen GP.

4.3 Requirements

Motivated by the above discussion, our aim is to develop an SDK and framework

that allows for the development and testing of standard-compliant TEE applications. The framework should allow development of GP-compliant CA and TA functionality without having to rely on any particular hardware support. Open-TEE is intended to be a fast prototyping and development environment that also provides a platform from which to conduct further research into TEE functionality. Our fundamental design principle is that it should require as little configuration and maintenance as possible, allowing the developer to focus on

the task at hand. We identify the following criteria by which we can measure our TEE framework's usefulness and hence its potential success in addressing the issues that motivated it.

- Compliance: Our framework should comply with GP's main interfaces, the GP TEE Client and GP TEE Core APIs.

- Hardware-independence: As a software based solution our framework should not be dependent on a particular TEE hardware environment. It should also not be dependent on any particular hardware for the development system itself.

- Reasonable performance: To be readily deployed, our framework must not suffer from code bloat that adds to the on-disk footprint nor to the memory consumption required to run it. In addition the start-up and restart times of the environment, especially that of the CAs and TAs should not be excessive. One of the perceived benefits of our framework is its ability to support fast prototyping and as such, any time penalty that is incurred will diminish its usability and the satisfaction of using it.

- Ease-of-use: The solution should be easily deployed and configured. It should use tools that are widely available making it more attractive (e.g., there should be no need for extra package/tool configuration on the development system

4.4 Architecture

The following section describes our design and implementation of such a software framework which we call 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.

4.4.1 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 as a whole. 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.

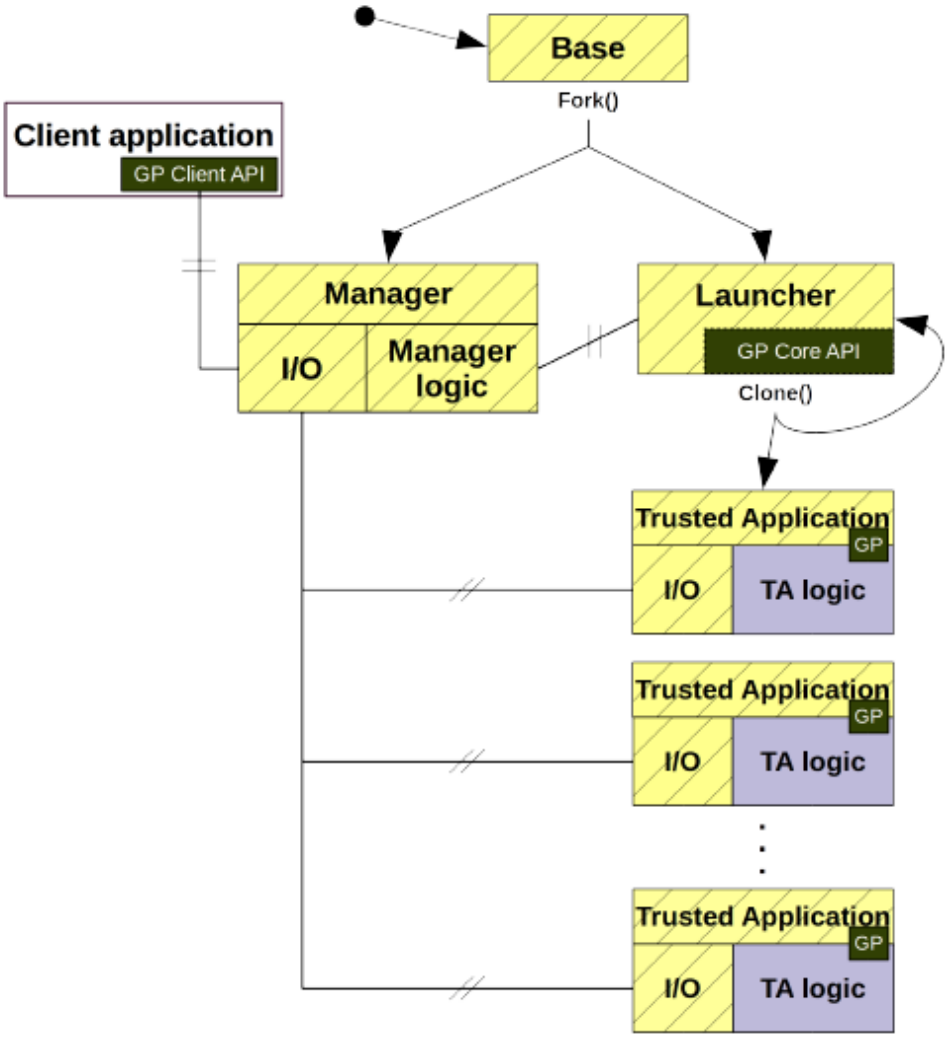


Figure 4.1

Open-TEE architecture

4.4.2 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.

4.4.3 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 [1]) 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 pre-loaded 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).

4.4.4 TA Processes

The architecture of the TA processes is inspired by the multi-process architecture utilized in the Chromium Project [38]. Each process has been divided into two threads3. 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 benefits of this model are that it allows greater separation and abstraction of the TA functionality from the Open-TEE framework.

4.4.5 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.

4.4.6 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.

4.5 Implementation and Tooling

This section highlights interesting points and background of the design, implementation and tooling choices.

4.5.1 Utilizing existing functionality

To meet the hardware-independence requirement, we do not emulate specific TEE hardware with software based emulators, such as QEMU [35]. Instead we rely on existing technologies and the services offered by the mainstream OS in which Open- TEE is running rather than developing a new TEE OS to deploy the GP APIsin. In addition we reuse software from existing open source projects, such as the OpenSSL4 crypto library and the GNU tool suite, thereby reducing the amount of time required to develop and test the Open-TEE framework.

This also contributes towards meeting the ease of use requirement in that developers can easily set up Open-TEE and start developing TAs using a set of familiar tools, editors, Integrated Development Environments (IDEs), compilers and debuggers. For example, a developer utilizing Open-TEE can connect to a TA process with a cheap reliable software debugger such as GDB [19] for detailed

debugging tasks like stepping through the code, inspecting variables and registers etc.

4.5.2 Development process

The intended user base for Open-TEE consists of seasoned developers. To ensure viability in such a demanding user base, we adopted a rigorous development process for Open-TEE so that the end result will be perceived as robust and usable. Open- TEE is developed as an open source project and as such there are a number of powerful tools that are freely available for this type of project. GitHub is used for hosting the code and GerritHub is used for performing peer-review of all code before it is submitted to the code base. In addition to the manual review process we leverage the power of Coverity to perform in depth static analysis scans. This enforces secure coding practices and helps to find potential functional bugs that may have been missed during the manual code review. In addition, we have deployed a Continuous Integration (CI) server running Jenkins8, which we have connected to GerritHub. Its main task is to perform a number of "smoke tests"9 on the new patches. These tests ensure that the patches conform to the coding guidelines, build successfully and that the basic system is usable after the patches are applied.

4.5.2 Open-TEE in use

Being designed as an open source framework upon which to build and test features that will utilize a TEE, Open-TEE has been implemented to be as inconspicuous as possible. The complexity of the system is hidden from the users of Open-TEE. They are presented with an SDK that exposes the GP TEE Client and GP TEE Core APIs without being required to have a deep understanding of how the overall framework works, thereby allowing them to focus on the development of their own TAs. However, Open-TEE is already being extended by the community. The ongoing implementation of the GP TEE Trusted User Interface (TUI) [16] 10 specification is an example.

4.5.3 Android and Open-TEE

During the Open-TEE specification phase it was recognized that there may be a demand for Open-TEE on a mobile device, because in general, most of the TEE use cases are associated with mobile devices and therefore it can be expected

that most of the developed TAs are designed for this domain. From a developer's perspective it would be valuable to test the application in the context of the device where it will be deployed, to ensure that e.g. communication from upper level through CA to TA is working correctly before deploying it to real hardware. Android was a natural target on mobile platforms for many reasons: it is widely deployed, open source and closely resembles GNU/Linux environment. Although Android was not the primary target, the Android aspect greatly affected the design of Open-TEE and implementation decisions, because the delta between the two environments should stay as minimal as possible in order to reduce the porting effort. To expand the example in the first paragraph, developers can test out their Java Native Interface (JNI) implementation with Open-TEE.

4.5.4 SGX and Open-TEE

SGX provides a TEE service through a set of CPU instructions which can be used for constructing an "enclav" to protect sensitive code and data. This has a number of subtle side-effects, e.g. the "enclavised" code cannot make any system calls or invoke services provided by the OS directly. In other words, the protected code cannot rely on any service framework. The code must be self-contained. But as discussed in Chapter 2, one of the benefits of having a standards compliant TEE is the possibility of using existing services. Developers could rely on the service framework provided by a TEE for e.g. accessing a keystore rather than maintaining the keystore by itself. Using SGX within Open-TEE has been a consideration from the beginning of the Open-TEE project. Utilizing SGX it should be possible to expand Open-TEE from a development aid into a fully functional TEE. In the course of writing this thesis, the efforts of hardening Open-TEE has been an ongoing effort, which has greatly affected Open-TEE design and implementation decisions.

For example, one of the reasons for separating the TA processes into an I/O and a logic thread is to enable the logic thread to be "enclavised" within SGX and offload the OS interaction to the I/O thread.

4.5.5 Fall back TEE

In general, securing Open-TEE is an intriguing topic, because modern OSs are equipped with various security mechanisms like Seccomp11 and LXC Containers. The future research work question might be formulated as: Could it be possible to combine various OS security mechanisms into one product and what level of

trust could be achieved by using these existing techniques and technologies? The edge of this approach is that the enabler technology is in place and it only needs to be utilized. In addition experiments could be conducted to see if these mechanisms are deployable in existing platforms and devices.

4.5.6 GlobalPlatform call for review

During our implementation of Open-TEE, GlobalPlatform announced an updated version of the GP TEE Core API. The implementation started with GP TEE Core version 1.0 and was nearly complete when GP announced a public review of the subsequent version. Under review was a working draft 1.0.26 and it received comments from all interested parties. This opportunity was utilized and the review of the working draft C was created in GP format and submitted to GP for public review.

The review is based on the experience of implementing Open-TEE:

- It raised the issue of some of the function descriptions. They may be difficult to understand or are ambiguous. For example the working draft function TEE_PopulateTransientObject description was ambiguous of how the function parameters should be handled e.g. should they be deep or shallow copied.

- Pointing out some of the unrealistic functional requirements. Some of the functionality might be questionable after evaluating the feasibility of implementing these requirements from the TEE vendor's point-of-view. An example of this is a static allocation of a cryptographic operation. The cryptographic operation is allocated with TEE_AllocateOperation function, which is the only function the cryptographic API subsection of the GP TEE Core API that can return the out of memory return code. Therefore if our implementation needs to be fully GP compliant, it must ensure that cryptographic operations are not failing due to allocation operations.

- Proposing new functionality. For example related to operation state; a new constant was proposed for improving the TEE Core API readibility and usability.

- Pointing out possible flaws in the working draft. For example Authenticated encryption state was incorrect after TEE_AEinit function, which would block subsequent calls e.g. TEE_AEUpdate.

- Suggested re-ordering the document layout with an aim to improve readability of the document.

GP reviewed the proposals and provided feedback of the submitted review. They made multiple changes on the suggestions in the review. For example the TEE_PopulateTransientObject function description wording was improved. Because the subsequent version of GP TEE Core API was a minor release, they did not accept any major changes. They were strict about not breaking the backward compatibility and therefore a couple of points from review are considered for the next major version.

4.5.7 GP Trusted User Interface (TUI)

TAs without the possibility to interact with the user severely limit the possible TEE/TA use cases. For example exchanging sensitive information with the user, i.e. when the user is entering authorization credentials (username, password) or the TA may want to display a calculated One-Time code to the user to allow them to log into a service. In such cases there is a need to interact with the TEE without the possibility of the REE intercepting the sensitive content. GP recognized this need and they have extended TEE specisification with TUI API [16]. The project builds on top of Open-TEE by implementing an extension, which provides the TUI functionality for TAs. One benefit of having a TUI framework with Open-TEE is that the developers are able to develop and test their TUI based TAs on the different platforms where Open-TEE is supported.

4.6 EVALUATION

The requirements are being continually evaluated, even as this thesis is being written, due to the ever evolving nature of Open-TEE. It is an active open source project to which more features are being added and existing features refined based upon the feedback that is received.

4.6.1 Compliance

Every effort has been made to comply with the GP standard. Whenever this has not been feasible, due to time constraints or in the interest of providing a platform upon which to build, the deviation has been documented and a debug message is logged to inform the user of the non-compliance. The GP TEE Client API is fully implemented. The GP TEE Core API implementation has 100%

function coverage, however, the algorithm coverage is currently 80% due to the use of existing libraries that do not support the remaining algorithms.

Information related to other implementations of the GP specification are scarce and most of the information related to it is proprietary and therefore it is not known if some of the commercial TEE implementations are fully GP compliant. If this were known, an idealistic validation of our implementation would be executing CAs and TAs we have created in an independently implemented TEE to compare the results.

A compliance test suite is commercially available from GP, however it is not freely available and needs to be purchased by non-affiliated members. Because Open-TEE is an open source project, it lacks the funds to purchase the use of this tool.

4.6.2 Hardware-Independence

By following the GP standard and not emulating any specific TEE hardware, Open-TEE is independent of TEE hardware. TAs developed with Open-TEE can be compiled to any target TEE hardware architecture. We have verified [13] that a non-trivial TA developed using Open-TEE (284 Lines of Code (LoC), 19 GP TEE Core API invocations (9 unique functions), 6 invokable TA commands) has been successfully compiled and run on a hardware TEE based on ARM TrustZone running the Trustonic <t-base environment [41].

Open-TEE can provide coverage reports to help highlight hot-spots in the code, generate call graphs etc. The GP TEE Core API includes memory management primitives and allows configuration parameters (such as gpd.ta.dataSize and gpd.ta.stackSize) to indicate how much heap and stack memory is available to a TA. A developer can use these parameters to configure Open-TEE to reflect the memory restrictions of a target hardware TEE environment.

However, as the actual TEE is potentially running a different environment than that offered by Open-TEE– possibly utilizing hardware based cryptographic accelerators, potentially having a different CPU, with different clock speed and throughput characteristics it will result in different timing characteristics. In this sense, as with all virtual environments, Open-TEE cannot fully replace the actual hardware environment for the final stages of the development cycle. Instead developers using Open-TEE can gain confidence that the hardware-independent parts of their trusted applications have been optimally implemented by making judicious use of coverage reports and other generic analysis techniques. Any hardware-specific optimization, such as performance tuning, naturally needs to be done on the target hardware environment.

Open-TEE has been deployed and used on various development environments ranging from servers to desktops and laptops1. It has been tested on both ARM and x86 architectures. Open-TEE requires Linux but has been run successfully on virtual machines hosted on other OSs. Having chosen not to emulate existing hardware to create the framework helps to ensure that the TAs created using it are portable as it is harder to create machine dependent code.

4.7 Open TEE Documentation

The goal of the Open-TEE open source project is to implement a "virtual TEE" compliant with the recent Global Platform TEE specifications .

Our primary motivation for the virtual TEE is to use it as a tool for developers of Trusted Applications and researchers interested in using TEEs or building new protocols and systems on top of it. Although hardware-based TEEs are ubiquitous in smartphones and tablets ordinary developers and researchers do not have access to it. While the emerging Global Platform specifications may change this situation in the future, a fully functional virtual TEE can help developers and researchers right away.

We intend that Trusted Applications developed using our virtual TEE can be compiled and run for any target that complies with the specifications.

The Open-TEE project is being led by the Secure Systems group as part of our activities at the Intel Collaborative Research Institute for Secure Computing

All activities of the project are public and all results are in the public domain. We welcome anyone interested to join us in contributing to the project.

4.7.1 Quick Setup Guide

Open-TEE requires qbs 1.4.2 or above. For Ubuntu 14.04 up-to-date packages of qbs are available from the qutIM PPA.

If requred start by installing the PPA for the qbs build system.

```
$ sudo add-apt-repository ppa:qutim/qutim

$ sudo apt-get update -y
```

For **Ubuntu 15.04** and above qbs is available in universal repositories. So no additional steps are required:

```
$ sudo apt-get install autoconf automake libtool uuid-dev libssl-dev libglu1-mesa-dev libelfg0-dev mesa-common-dev build-essential git curl htop pkg-config qbs gdb libfuse-dev
```

Introduce yourself to `git`.

```
$ git config --global user.name "FIRST SECOND"

$ git config --global user.email "name@email.com
```

Now configure `qbs`:

```
$ qbs setup-toolchains --detect

$ qbs config defaultProfile gcc
```

Fetch the `repo` repository management tool:

```
$ mkdir -p ~/bin

$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo

$ chmod +x ~/bin/repo
```

Create a directory where to checkout the Open-TEE repositories:

```
$ mkdir Open-TEE

$ cd Open-TEE
```

Have `repo` fetch the manifest for the Open-TEE project:

```
$ ~/bin/repo init -u https://github.com/Open-TEE/manifest.git

$ ~/bin/repo sync -j10
```

Open the configuration file with your preferred editor:

```
$ sudo $EDITOR /etc/opentee.conf
```

Add the sample configuration given below to the configuration file (IMPORTANT DO NOT REMOVE [PATHS] section header):

```
[PATHS]
```

```
ta_dir_path = <PATHNAME>/Open-TEE/gcc-debug/TAs

core_lib_path = <PATHNAME>/Open-TEE/gcc-debug

opentee_bin = <PATHNAME>/Open-TEE/gcc-debug/opentee-engine

subprocess_manager = libManagerApi.so

subprocess_launcher = libLauncherApi.so
```

where `<PATHNAME>` with the absolute path to the parent directory of the Open-TEE directory you created earlier. The pathname must **not** include special variables such as `~` or `$HOME`.

Finally change working directory to where you cloned the sources, build Open-TEE and launch the `opentee-engine`:

```
$ qbs debug

$ ./opentee start
```

Verify that Open-TEE is running with `ps`:

```
$ ps waux | grep tee
```

You should see output similar to the example below:

gcc-debug$ ps waux |grep tee
brian 5738 0.0 0.0 97176 852 ? Sl 10:40 0:00 tee_manager
brian 5739 0.0 0.0 25216 1144 ? S 10:40 0:00 tee_launcher
If you do not see the 2 tee_ processes, open syslog in another terminal to see any errors:

```
$ tail -f /var/log/syslog
```

In the main terminal run a client test application:

```
$ gcc-debug/conn_test_app
```

You should now expect to see output similar to the following:

Open-TEE$ gcc-debug/conn_test_app
START: conn test app
Initializing context: initialized
… END: conn test app

*!!! SUCCESS !!! Connection test app did not found any errors. ^^^ SUCCESS ^^^*

4.7.2QBS

For QBS installation instructions see the Quick Setup Guide.

Configure `qbs` for your toolchain:

```
$ qbs detect-toolchains
$ qbs config --list profiles
```

Optionally you may select one of the profiles to be the default one e.g. to set the `gcc` profile as default

```
$ qbs config defaultProfile gcc
```

Finally, build Open-TEE:

```
$ qbs debug
```

The result of the compilation will be found under `<profile>-debug`, e.g. executables and libraries under `gcc-debug` and trusted application objects under `gcc-debug/TAs`.

4.7.3 Autotools

The Autotools build has been tested with Autoconfig 2.69 and above. To perform an Autotools build you need to install `autoconf`, `automake` and `libtool`:

```
$ sudo apt-get install autoconf automake libtool
```

4.7.4 Building with Autotools

We recommend using a parallel build tree (a.k.a. `VPATH` build):

```
$ mkdir build
```

The provided `autogen.sh` script will generate and run the `configure` script.

```
$ cd build
$ ../autogen.sh
```

To build and install Open-TEE run:

```
$ make
```

```
$ sudo make install
```

By default Open-TEE will be installed under `/opt/Open-TEE`. The directory will contain the following subdirectories:

- `/opt/Open-TEE/bin` - executables
- `/opt/Open-TEE/include` - public header files
- `/opt/Open-TEE/lib` - shared library objects (*libdir*)
- `/opt/Open-TEE/lib/TAs` - trusted application objects (*tadir*)

4.7.5 Configure Runtime Environment

Open the configuration file with your preferred editor:

```
$ sudo $EDITOR /etc/opentee.conf
```

Add the sample configuration given below to the configuration file:

```
[PATHS]
ta_dir_path = <PATH_TO_TA_DIR>
core_lib_path = <PATH_TO_LIB_DIR>
opentee_bin = <PATH_TO_BINARY>
subprocess_manager = libManagerApi.so
subprocess_launcher = libLauncherApi.so
```

where `<PATH_*>` is the absolute path to the directory of the directory created earlier. The pathname must **not** include special variables such as `~` or `$HOME`.

For a `qbs` build you can use:

```
[PATHS]
ta_dir_path = <PATHNAME>/Open-TEE/gcc-debug/TAs
core_lib_path = <PATHNAME>/Open-TEE/gcc-debug
opentee_bin = <PATHNAME>/Open-TEE/gcc-debug/opentee-engine
subprocess_manager = libManagerApi.so
```

```
subprocess_launcher = libLauncherApi.so
```

Where <PATHNAME> is replaced with the absolute path to the parent directory of the Open-TEE directory you created earlier. Yet again the pathname must **not** include special variables such as ~ or $HOME.

For an autotools build you can use

```
[PATHS]

ta_dir_path = /opt/Open-TEE/lib/TAs

core_lib_path = /opt/Open-TEE/lib

opentee_bin = /opt/Open-TEE/bin/opentee-engine

subprocess_manager = libManagerApi.so

subprocess_launcher = libLauncherApi.so
```

4.7.6 Running from the command line

We recommend adding the opentee script to your $PATH

```
$ cd ~/bin

$ ln -s <PATH_OPEN_TEE_REPO>/project/opentee opentee
```

Now you can start, stop, restart opentee more freely

```
$ opentee start

$ opentee stop

$ opentee restart
```

If you see either of the following errors, make sure you have configured /etc/opentee.conf as described in Configure Runtime Environment, paying particular attention to where the opentee binary is being built:

No conf file exists

Could not find binary name for opentee

To run the sample CA binaries navigate to the build directory for your choosen make tool e.g.

```
$ cd <PATH_TO_OPENTEE>/gcc-debug
```

```
$ ./conn_test_app
```

If the corresponding TA is not running then the Open-TEE framework will ensure that it is loaded as part of the TEEC_OpenSession() command from the CA.

4.7.7 Debugging with GDB

This is a quick start guide on how to debug our CA and TA applications. If you are similar with GDB debugger, the interesting part is how attach to TA process. This guide uses our connection test application (CA: conn_test_app ; TA: ta_conn_test_app) as an example.

4.7.8 Pre-setup of GDB

Ptracing of non-child process by non-root user is disabled by default in Ubuntu. In other words only root can ptrace every process and non-root can only ptrace its own child process. You might bump into this problem when you are trying to attach to running process and you might get following error:

```
Could not attach to process. If your uid matches the uid of the target

process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or

try again as the root user. For more details,

see /etc/sysctl.d/10-ptrace.conf ptrace: Operation not permitted.
```

You can solve error temporarily by disabling the restriction:

```
$ echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

or you can disable it permanently by editing /etc/sysctl.d/10-ptrace.conf

```
kernel.yama.ptrace_scope = 0
```

4.7.9 Debugging CA process

Navigate into our CA application binary folder and launch GDB with our CA name as a command line parameter:

```
 $ gdb conn_test_app
```

Set as many breakpoints as you require eg.:

```
$ break <OUR FUNCTION NAME>

$ break <SOURCE FILE NAME:LINENUMBER>
```

```
(for connection test application)

$ break full_treatment_test

$ break ta_conn_test_app.c:339
```

Run our CA process in GDB by hitting “r”

4.7.10 Debugging TA process

Debugging TA processes are generally done in the same way as debuggin our CA processes. You will be setting eg. break points and inspecting memory locations same way as in CA process.

TA processes are managed by Open-TEE framework and one of its many responsibilities are launching new TA processes. The inner workflow of launching new TA process is that it is beginning from Open-TEE manager process, which will be noticing that new TA process is need and therefore it will be communicating to Open-TEE laucnher process. Launcher will launch new TA process by forking it self. One process stays a launcher process and another one becomes a new TA process.

4.7.11 Debugging from the beginnings

This method is working for all TAs regardless of TA type. With this debugging method you can debug TA_CreateEntryPoint and TA_OpenSessionEntryPoint functions. The setup for this method is that our TA is not yet launched by Open-TEE framework. Our TA is launched by tee_launcher -process and therefore we need attach GDB to tee_launcher process. Attach to launcher process:

```
$ gdb gcc-debug/opentee-engine `pgrep tee_launcher`
```

Set GDB to follow child, because the new TA will be a child process of launcher

```
$ set follow-fork-mode child
```

Before hitting “c” for continuing GDB executing, you may set our TA process break points eg.:

```
$ break TA_CreateEntryPoint
```

If GDB is prompting following messaga, just select “y”

```
Function "TA_CreateEntryPoint" not defined.

Make breakpoint pending on future shared library load? (y or [n])
```

Run our corresponding CA application to get our TA running.

4.7.12 Debugging keep alive TA

This method of debugging is only working if our TA is set to be “keep alive”, which means that the TA is not getting destroyed, if there are no active connections to that TA. This method requires less steps and therefore could be a bit faster (do not have to set follow fork mode). This method only needs our TA PID:

```
$ gdb 'pgrep -f lib<OUR_TA_NAME>.so'


(for connection test application)

$ gdb 'pgrep -f libta_conn_test_app.so'
```

or find out our TA process PID manually and then attach GDB:

```
$ ps waux | grep lib<OUR_TA_NAME>.so

$ gdb attach <PID>
```

Now you are debugging a TA process. You may set a break point and when you are finished with the break points, continue GDB execution by hitting “c”

4.8 Android Build

This page has instructions on how to build Open-TEE for android devices. Currently tested only on android 5.1+ .

4.8.1 Quick Setup Guide

Start by following the instructions here to setup your android build environment and download the android source code (e.g. to $HOME/android_source).

Add the following to your .bashrc file or to a file you will source on each shell.

```
export ANDROID_ROOT="$HOME/android_source"

export USE_CCACHE=1

export CCACHE_DIR="$HOME/android_source/.ccache"

$ANDROID_ROOT/prebuilts/misc/linux-x86/ccache/ccache -M 50G
```

```
source "$HOME/android_source/build/envsetup.sh"
```

Inside $ANDROID_ROOT create a link to the directory you have downloaded Open-TEE to (e.g. $HOME/Open-TEE) like so:

```
ln -s $HOME/Open-TEE $ANDROID_ROOT/Open-TEE
```

Now to build just run

```
lunch
```

to choose the target and then

```
make clean && make opentee-engine libManagerApi libInternalApi libLauncherApi libCommonApi libta_conn_test_app conn_test_app libtee
```

to build all the Open-TEE modules. You can find all the available modules by doing `grep -ir "LOCAL_MODULE " Open-TEE/` where Open-TEE/ is the directory containing the Open-TEE source code.

The output files will by default be located in `$ANDROID_ROOT/out/target/product/generic\*/` (depending on the architecture). You should also be able to see the output directory path if you do `echo $OUT`.

To deploy those binary files to an Android device you can choose one of two methods:

###ADB (needs root on device)

To copy those files to an android device you can use the script located in Open-TEE/project/install_android.sh . The script assumes the `adb` binary is in your $PATH and that $OUT contains the directory where the binaries were outputted.

Note: root access on the device is needed for this.

The files should now be installed on */system/lib/{ta,tee}* and */system/bin* on the device. Do `adb root` to start adb with root privileges and then `adb shell` to get a shell on the device.

From the `adb shell`:

In case the files do not have an execution permission add it with something like:

```
chmod +x /system/bin/opentee-engine

chmod +x /system/bin/conn_test_app
```

And run Open-TEE with

```
/system/bin/opentee-engine
```

Verify that Open-TEE is running with `ps`:

```
ps | grep tee
```

####Importing and testing a TA via adb

Modify *Open-TEE/project/install_android.sh* to also copy your TA .so file to the */system/lib/ta/* directory and your CA to */system/bin/* Run your CA to test the TA directly via adb shell with /system/bin/ and check logcat or gdb on android for debugging output.

###Android Studio

This method uses an Android Studio project that packages Open-TEE inside an application and installs/runs it to the home directory of the app.

Start by cloning the repo with

```
git clone https://github.com/Open-TEE/opentee-android

cd opentee-android/
```

Then import all the binaries built to the opentee_mainapp module (that packages Open-TEE) by using the *opentee_mainapp/install_opentee_files.sh*. For each architecture compiled (armeabi, armeabi-v7a, x86) you should re-run the install_opentee_files.sh with the appropriate argument (do `./install_opentee_files.sh -h` for help).

After the import and assumming you have downloaded and installed Android Studio use it to open the opentee-android project (File/Open…). You might need to specify the Android NDK or Android SDK directory in *local.properties* but the IDE should in most cases detect those by itself.

You can then build and run the testapp module that is a reference usage implementation and demonstrates how to install/run Open-TEE and other binaries. For more information on how to use opentee-android check the README.md included in the project.

4.8.2 Troubleshooting

If you get errors similar to:

```
D/tee_manager(32036): opentee/emulator/opentee-main/main.c:load_lib:166 Failed to load library, /system/lib/libManagerApi.so : dlopen failed: cannot locate symbol "mempcpy" referenced by "libCommonApi.so"...

D/tee_launcher(32037): opentee/emulator/opentee-main/main.c:load_lib:166 Failed to load library, /system/lib/libLauncherApi.so : dlopen failed: cannot locate symbol "mempcpy" referenced by "libCommonApi.so"...
```

Then most probably the android tree that you are building with does not match the tree on the device and thus you might also have to push the generated libc.so (or other lib*.so files) to the device.

Note: that this is unsafe and might result in your device malfunctioning. It is a good idea to take a backup of the /system/lib/lib.so files or even a complete ROM backup if you have a custom recovery.*

If you get errors similar to

```
error: unknown target 'opentee-engine'
```

or for another Open-TEE module then consider copying the $HOME/Open-TEE directory directly under the $HOME/Open-TEE $ANDROID_ROOT/ tree instead of symlinking since that might be causing the issues.

5. Conclusions

Open-TEE was initially intended to be a developer tool. Although use of TEEs can improve the security and usability of their service, not all their clients may have

TEE-equipped devices. Yet the service provider would like to present a consistent user experience for their entire client base. A possible approach for them is to ship their application (CA and TA) with Open-TEE and arrange for the CA to use Open-TEE if it cannot detect a real hardware TEE on the device. This would allow the service provider to have a common provisioning mechanism and offer a consistent user experience for all their clients. However, once Open-TEE has been cast as a potential fall-back TEE in this manner, there emerges the need to address the question of how it would be best to isolate it from the REE in the absence of any hardware support. Reiterating that Open-TEE is not intended to emulate any specific TEE hardware. Open-TEE meets its goal of guaranteeing that trusted applications developed using it will compile and run on any GP-compliant TEE hardware. Hardware-specific aspects, such as performance tuning are outside the scope of Open-TEE. It is believed that organizations and developers who already develop TA applications will benefit from incorporating Open-TEE into their development process.

References

[1] Open-TEE - An Open Virtual Trusted Execution Environment, Brian McGillion∗ , Tanel Dettenborn† , Thomas Nyman‡ Intel Collaborative Research Institute for Secure Computing (ICRI-SC), N. Asokan§ at Aalto University, Finland

[2]Apple, "iOS security," https://www.apple.com/ca/iphone/business/docs/iOS_Security_Feb14.pdf.

[3] ARM, "Technical reference manual: ARM 1176jzf-s (trustzone-enabled processor),"
http://www.arm.com/pdfs/DDI0301D_arm1176jzfs_r0p2_trm.pdf.

[4] ARM, "ARM security technology — Building a secure system using Trust-Zone technology," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html, April 2009.

[5] J. Azema and G. Fayad, "M-Shield mobile security technology," 2008, TI White paper. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.

[6] D. Balfanz and E. W. Felten, "Hand-held computers can be better smart cards," in Proceedings of the 8th USENIX Security Symposium, Washington, D.C., August 23-26, 1999, 1999. [Online]. Available: https://www.usenix.org/conference/8th-usenix-security-symposium/hand-held-computers-can-be-better-smart-cards

[7] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability," International Journal of Human-Computer Interaction, pp. 574–594, 2008, http://dx.doi.org/10.1080%2F10447310802205776.
[8] J. Brooke, Usability evaluation in industry. Taylor & Francis, London, 1996, ch. SUS: A "quick and dirty" usability scale, pp. 189–194.

[9] Y. Cheng, X. Ding, and R. Deng, "Appshield: Protecting applications against untrusted operating system," Singaport Management University Technical Report, SMU-SIS-13, vol. 101, 2013.

[10] Common Vulnerabilities and Exposures (CVE), "Cve-2015-3456," https://cve.
mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456.

[11] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 secure coprocessor,"

[12] J.-E. Ekberg, "Securing software architectures for trusted processor environments,"
Doctoral dissertation, Aalto University, May 2013, http://urn.fi/URN:ISBN:978-952-60-3632-8.

[13] J.-E. Ekberg, "Personal communication," 2015, Trustonic.

[14] J. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential

of trusted execution environments on mobile devices," IEEE Security & Privacy, vol. 12, no. 4, pp. 29–37, 2014. [Online]. Available: http://dx.doi.org/10.1109/MSP.2014.38

[15] GlobalPlatform, "About." http://www.globalplatform.org/aboutus.

[16] GlobalPlatform, "Device specifications for trusted execution environment." http://www.globalplatform.org/specificationsdevice.asp.

[17] GlobalPlatform, "Home page." http://www.globalplatform.org.

[18] GlobalPlatform, "TEE System Architecture," http://www.globalplatform.org/ specificationsdevice.asp.

[19] GNU, "GDB: The GNU project debugger," http://www.gnu.org/software/ gdb/.

[20] GNU, "General public license," https://gnu.org/licenses/gpl.html.

[21] Intel, "Intel software guard extensions (intel sgx)," https://software.intel.com/en-us/intel-isa-extensions#pid-19539-1495.

[22] Intel, "SEP driver," https://git.kernel.org/cgit/linux/kernel/git/stable/ linux-stable.git/tree/drivers/staging/sep?id=refs/tags/v3.14.32.

[23] Intel, "Software guard extensions programming reference," https://software.intel.com/sites/default/files/329298-001.pdf.

[24] K. Kostiainen, J. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009, 2009, pp. 104–115. [Online]. Available: http://doi.acm.org/10.1145/1533057.1533074

[25] K. Kostiainen, E. Reshetova, J.-E. Ekberg, and N. Asokan, "Old, new, borrowed,blue–: a perspective on the evolution of mobile platform security architectures,"in Proceedings of the first ACM conference on Data and applicationsecurity and privacy. ACM, 2011, pp. 13–24.

[26] M. Leno, "Senate bill 962, leno. smartphones." http://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201320140S962.

[27] Linaro, "OP-TEE," https://wiki.linaro.org/WorkingGroups/Security/ OP-TEE.

[28] 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

[29] F. McKeen et al., “Innovative instructions and software model for isolated execution,” in Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy, ser. HASP ’13. New York, NY, USA: ACM, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[30] A. Muthu, “Emulating trust zone feature in android emulator by extending qemu,” Master’s thesis, KTH Royal Institute of Technology, 2013.

[31] A. Muthu, R. Rahmani, and D. Rajaram, “Emulating trust zone in android emulator with secure channeling,” International Journal of Computer Science Issues, vol. 10, no. 5, pp. 40–51, 2013.

[32] R. Needham and A. Herbert, “The cambridge cap computer and its operating system,” 1982.

[33] NVIDIA, “Trusted little kernel (tlk),” http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=summary.

[34] Official California Legislative Information, “Senate bill no. 962,” http://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201320140SB962.

[35] QEMU, “Open source processor emulator,”http://wiki.qemu.org/Main_Page.

[36] Sierraware, “Open virtualization’s SierraVisor and SierraTEE,” http://www.openvirtualization.org/.

[37] The Apache Software Foundation, “Apache license, version 2.0,” http://www.apache.org/licenses/LICENSE-2.0.

[38] The Chromium Projects, “Multi-process architecture,” http://www.chromium.org/developers/design-documents/multi-process-architecture.

[39] “Trusted Platform Module (TPM) Specifications,” https://www.trustedcomputinggroup.org/specs/TPM/.

[40] TrustKernel, “T6,” http://trustkernel.org/.

[41] Trustonic, “<t-dev developer program,” https://www.trustonic.com/products-services/developer-program/.

[42] J. Winter, P. Wiegele, M. Pirker, and R. Tögl, “A flexible software development and emulation framework for ARM TrustZone,” in Trusted Systems - Third International Conference, INTRUST 2011, Beijing, China,

November 27-29, 2011, Revised Selected Papers, 2011, pp. 1–15.