Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Rui Yang

Java APIs for Trusted Execution Environments

Master's Thesis
Espoo, June 16, 2016

Supervisors: Professor N. Asokan, Aalto University
Associate Professor Frank Alexander Kraemer, Norwegian University of Science and Technology
Instructor: Andrew Paverd Ph.D
Thomas Nyman M.Sc

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Rui Yang

Title:
Java APIs for Trusted Execution Environments

Date:	June 16, 2016	**Pages:**	94
Professorship:	Security and Mobile Computing	**Code:**	T-110
Supervisors:	Professor N. Asokan Associate Professor Frank Alexander Kraemer		
Instructor:	Andrew Paverd Ph.D Thomas Nyman M.Sc		

Based on GlobalPlatform (GP) Trusted Execution Environment (TEE) specifications, Open-TEE paved the way for ordinary developers to create and deploy Trusted Applications in a GP-compliant TEE. However,when developing an Android Client Application which intends to use the functionality of the GP-Compliant TEE, there still lacks an easy way of using the C binding GP TEE Client API. In this thesis, the problem is addressed in more details and the proposed solution by designing and prototyping a Java API is discussed.

Keywords: TEE, GlobalPlatform, Open-TEE, Java API

Language: English

Acknowledgements

I wish to thank all the people who have helped me to finish this master thesis especially my two supervisors and two advisors.

Hope you all the best.

Espoo, June 16, 2016

Rui Yang

Abbreviations and Acronyms

TEE	Trusted Execution Environment
REE	Rich Execution Environment
TAs	Trusted Applications
CAs	Client Applications
GP	GlobalPlatform [10]
OSs	Operating Systems
API	Application Program Interface
OEM	Original Equipment Manufacturer
NDK	Android Native Development Kit
SoC	System on Chip
SGX	Software Guard Extensions
ObC	On-board Credentials
SMC	Secure Monitor Call
ELF	Executable and Linkable Format
JNI	Java Native Interface
UUID	Universally unique identifier
RMR	Registered Memory Reference
JVM	Java Virtual Machine
IPC	Inter-process Communication
PID	Process Identifier
PCs	Personal Computers
TPM	Trusted Platform Module
TCG	Trusted Computing Group
TSS	TCG Software Stack
TSP	Trusted Service Provider
TCS	Trusted Core Service
TDDL	Trusted Device Driver Library
TSPI	TSP Interface
MAC	Mandatory Access Control

DAC	Discretionary Access Control

Contents

Chapter 1

Introduction

For the last decade, the vast majority of mobile devices are shipped with a hardware-based *Trusted Execution Environment* (TEE) [6]. A TEE is an isolated computational environment which provides integrity protection and secure storage services to the outside untrusted world. It is separated from the *Rich Execution Environment* (REE) in which normal operating systems (OSs) are running. By keeping confidential information and limiting its manipulations within the TEE, even if the REE is compromised, it is still hard for critical information breaches from the TEE. A general TEE consists of a hardware trust anchor, such as the ARM TrustZone [3], and software including a Trusted OS. It is possible to attack the software parts. One example can be found in the exploit CVE-2015-6639 [1]. However, the capacity of the TEE is much smaller than the REE and it can only allow limited operations, which makes the TEE more manageable and susceptible to fewer bugs. Moreover, the TEE normally exposes its service to the outside world using a component defined in the REE, such as a kernel driver. Client Applications (CAs) running inside the REE can utilize the functionality of the TEE via an Application Program Interface (API) exported via this component. Nowadays, ordinary developers do not have access to these APIs which are limited to TEE venders, Original Equipment Manufacturer (OEM) and service providers who want to use the TEE to protect their digital properties, e.g. Netflix. The lack of standardized API is also a big challenge for ordinary developers to benefit from the TEE. To remove such an obstacle, the GlobalPlatform(GP) came to play the role by publishing the GP TEE Client API specification [7] which standardizes the way how CAs can interact with a GP-Compliant TEE. A recent promising implementation of GP TEE related specifications can be found in Open-TEE [22] which is a virtual TEE

[1] QSEE privilege escalation vulnerability and exploit, `https://bits-please.blogspot.dk/2016/05/qsee-privilege-escalation-vulnerability.html`

that can provide development environments for CAs and Trusted Applications (TAs). A prototype implementation which includes our solution is also based on it.

Fact is that the GP TEE Client API comes with a form of C style. Fact is, too, that the mainstream programming language of Android application development is Java. This poses a challenge for Android developers who wish to utilize the functionality of a GP-compliant TEE. Although thanks to the Android Native Development Kit (NDK) [12], developers can write native code in their applications using the GP TEE Client API to interact with a GP-compliant TEE. It is still difficult to deal with complex native code especially for those who are not familiar with it.

So how can Java developers use the GP-compliant TEEs directly from Java? The first challenge that we are faced with is how different design principles specified in GP TEE Client API, such as the shared memory between CAs and TAs, should be transformed while still function in an acceptable manner. Another challenge is how to expose the data types to ordinal Java developers in a way that the underlying implementations can be hidden from them and also provide flexibility for different kinds of underlying implementations. To overcome these two challenges, we have introduced a Java interface for TEEs. **The ultimate goal is to allow Java developers directly access a GP-compliant TEE.** GP TEE Client API enables developers to access to the GP-compliant TEE using C API. So a Java wrapper for C API is needed. As a result, we have mapped the GP TEE Client API to a Java API named with *OT-J*. Instead of directly replacing the data types defined in GP TEE Client API to Java types, we have converted all C structs to Java interfaces and all C API are embedded into corresponding Java interfaces. Benefiting from the information hiding feature of these Java interfaces, ordinal Java developers need not to worry about the underlying implementations and it is also possible for alternative implementations of the *OT-J*.

In this thesis, we target a Java API on Android.

The contributions of this thesis are:

1. Designing a GP TEE Client Java API named as OT-J (its documentation presented in Appdendix A) and describing the design philosophy behind it (Chapter 4);

2. A prototype implementation in Android, which fully realize OT-J (Chapter 5);

3. Evaluating OT-J using existing GP-Compliant TEE and TA (Chapter 6).

Chapter 2

Background

This chapter describes the background information related to this thesis.

2.1 TEE & REE

As stated in the previous chapter, TEE is an isolated computing environment which can offer the benefit of platform boot integrity, device identification/authentication, integrity protection and secure storage services for the REE [6]. See the TEE general architecture in Figure 2.1. The conventional REE represents an operating system (REE OS), such as Windows, Mac OS X, Linux, Android or iOS. It abstracts the underlying hardware and provides corresponding resources and functionality for the CAs to run within/top of it. Normally it has rich features compared to the TEE OS. However, the REE is vulnerable to different kinds of attacks, such as malware. In order to protect sensitive private information, such as decryption private keys, against these attacks, it is good to keep this information safely in a separate container in case the REE is compromised. The TEE provides the isolation for such a secure container. In addition, a small set of applications named TAs can run within the TEE. The combination of secure storage and constrained operations on it defined by the TAs can keep the private information securely inside of the TEE without exposing it to the REE.

The TEE is isolated from the REE by using either a security co-processor or processor secure environment [8]. Normally the REE is integrated into a physic chip named *System on Chip* (SoC). In the first case, the security co-processor can be a processor either inside or outside of the SoC module. The TEE OS runs on top of the security co-processor along with the REE OS running in a normal processor. As such, the TEE and REE are separated from each other by strict hardware boundaries. For the device only equipped

with one processor, another architecture for the TEE is providing a processor secure environment on this single processor with the REE. So the TEE and REE share common resources but only one of them can function at a given time. This approach only requires one processor which is cheaper to deploy but lacks of efficiency compared with the first one.

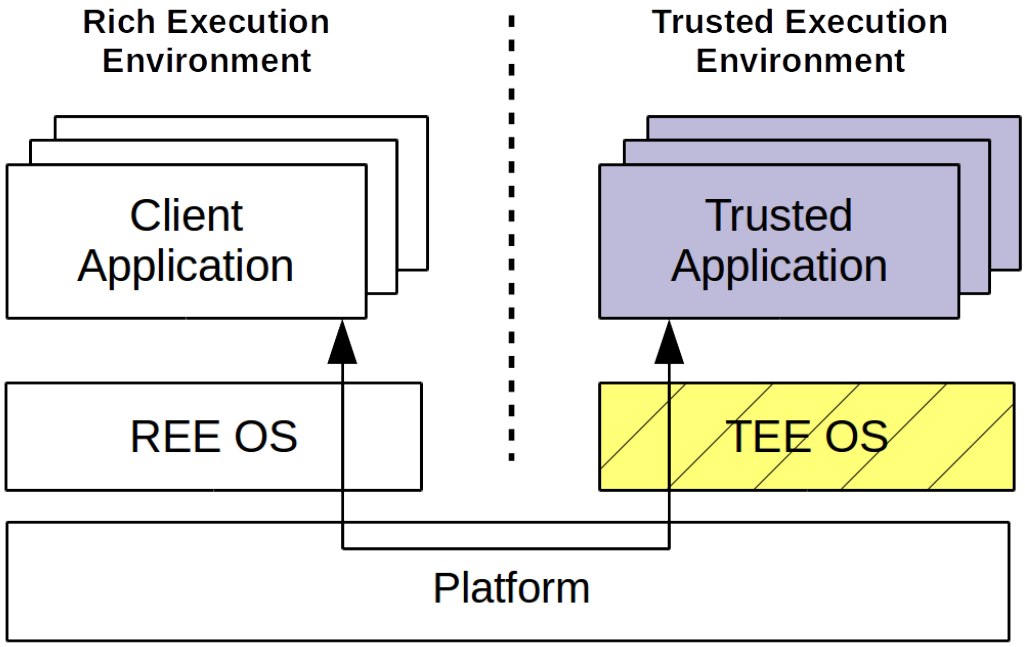

Figure 2.1: TEE general architecture [22]

2.2 ARM TrustZone

ARM TrustZone [2] is one of the outcomes by applying the second approach as stated in previous section. In TrustZone, the TEE is regarded as the *secure world* and the REE as the *normal world*. To establish a reliable trust model, the boot procedure of the device starts at the processor loading the TEE OS and running it in a secure world. Once the TEE OS booted up, it will signal the processor to switch to normal world and begin booting the REE OS. After that, the REE OS can make a Secure Monitor Call (SMC) to allow the TEE OS control the processor again. In this case, the processor has to switch the context back and forth between the secure and normal world. Because they share the common resources, when the processor changed to

normal world, the storage of the TEE is protected using encryptions and only the TEE can access to it. Due to the fact that only one world can remain in the processor, this helps to isolate the TEE and REE by disabling their direct interactions. However, context switches of the processor come with a big overhead which is the major drawback with this solution.

2.3 Intel Software Guard Extensions (SGX)

SGX is a software extension and hardware-based approach provided by Intel [18, 23], which also adopts the second approach by creating a processor secure environment to create a TEE. Unlike the context switches between the secure and normal world in TrustZone, there is no such switch in the SGX. The processor which runs SGX holds a private key limited to the CPU which is used to decrypt part of an application called *Enclave*. The Enclave is the encrypted code and data which can be decrypted only by the CPU. When the Enclave data leaves the CPU, it will be encrypted again. By limiting the usage of the Enclave within the CPU, it can prevent external accesses. Compared with TrustZone, without the complex context switch, SGX does not need to restore the previous device state.

2.4 Use cases for TEE

This section states the efforts which adopt the TEE to other solutions.

On-board Credentials (ObC) Credentials are widely used to protect communications via insecure channels or for the purpose of authentications. ObC [20] is an architecture to secure these credentials using general secure hardware, such as the TEE. It consists of the following components:

1. *ObC Interpreter* is a simplified virtual machine running inside an isolated environment provide by secure hardware, which only allows one program run at time due to the limited resources on targeting secure hardware. The program can be deployed by any developers;

2. *Credential Manager* is a process running outside of secure environment and interacts with other ObC components, which exposes the ObC service to CAs;

3. *ObC Database* also resides outside the secure hardware, which stores the encrypted credentials;

4. *ObC Provisioning System* is a system running inside secure environment which provisions the installation and usages for credentials.

One example of adopting the TEE secure hardware using TrustZone into the ObC architecture can seen in Figure 2.2.

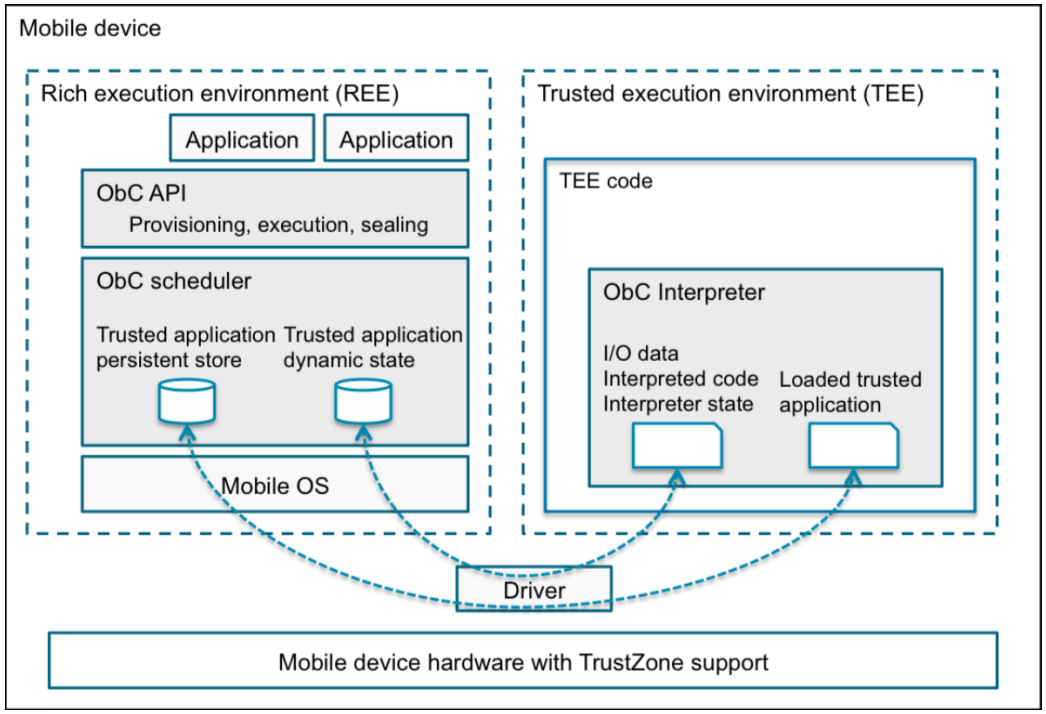

Figure 2.2: ObC Architecture on mobile device [6]

ObC interpreter runs inside the TEE. Depending on specific TrustZone implementations, ObC can be regarded either a simplified TEE OS or just a normal TA. It isolates the program running inside of it from the resources provided by the TEE. ObC scheduler at the bottom left of the figure provides consistent storage for encrypted credentials similar to ObC Database. It also triggers the interrupt between different programs based on the signal from the upper layer ObC API call. The functionality of such a ObC system, including provisioning, execution and sealing, is exported by ObC API.

TEE-backed KeyStore Since Android 4.3 (API level 18), Android provides a secure container named *KeyStore*. The KeyStore can keep cryptographic keys away from malicious extractions and unauthorized usages [11]. When a TEE is present in an Android device, the KeyStore system will use

the TEE to store the keys instead of using a software implementation. The TEE in here mainly stands for the TEE provided by vendors which does not share common API interfaces. The TEE defined in GP can achieve the same functionality as the KeyStore system. First of all, the keys stored in GP-compliant TEE are also tamper-proof from the untrusted outside world excluding valid CAs. Moreover, the feature of key usage authentications is provided during opening a session to a TA in which the CA must validate itself to the TA by providing correct authentication data. However, the API provided by GP TEE Client API opens a wider implementation possibility.

Linux generic TEE subsystem is a TEE driver available for the GP-compliant TEE, which is released in a Linux kernel patch [19]. It facilitates communication between the TEE and REE. Via the normal usage of a Linux driver, such as *open*, *ioctl* and *close*, the untrusted world can implement the GP TEE Client API based on these calls.

2.5 GlobalPlatform (GP) TEE Client Specification

GP is a non-profit organization that publishes specifications to promote security and interoperability of secure devices. One of the specifications it publishes, the "GlobalPlatform Device Technology TEE Client API Specification" (GP TEE Client API) [7], standardizes the ways in which CAs communicate with the TEE and TAs. All its APIs are showed in Table 2.1. And data structures are presented in Listing 2.1. GP also has other specifications aimed for TEEs but we only focus on this one specifically since it is the foundation of our Java API. This specification defines the C data types and API functions for CAs to utilize. The C style specification can provide more rich information compared with other programming languages. In addition, most low level components are implemented using C, for instance Linux drivers. So C data types and APIs are directly used to dealing with such components which can be more compatible. Since how to deploy TAs and extra access controls are not mentioned in GP TEE Client API, they are also omitted from the OT-J.

TEEC_InitializeContext
TEEC_FinalizeContext
TEEC_RegisterSharedMemory
TEEC_AllocateSharedMemory
TEEC_ReleaseSharedMemory
TEEC_OpenSession
TEEC_CloseSession
TEEC_InvokeCommand
TEEC_RequestCancellation

Table 2.1: GP TEE Client API

```
typedef struct
{
    void* buffer;
    size_t size;
    uint32_t flags;
    <Implementation-Defined Type> imp;
} TEEC_SharedMemory;

typedef struct
{
    TEEC_SharedMemory* parent;
    size_t             size;
    size_t             offset;
} TEEC_RegisteredMemoryReference;

typedef struct
{
    uint32_t a;
    uint32_t b;
} TEEC_Value;

typedef union
{
    TEEC_TempMemoryReference        tmpref;
    TEEC_RegisteredMemoryReference  memref;
    TEEC_Value                      value;
} TEEC_Parameter;

typedef struct
{
    uint32_t     started;
    uint32_t     paramTypes;
    TEEC_Parameter  params[4];
    <Implementation-Defined Type> imp;
```

```
35 } TEEC_Operation;
```

Listing 2.1: GP data types [7]

2.6 Open-TEE

Open-TEE [22] is an open-source virtual TEE which is a software implementation based on the GP TEE Specifications with no dependency on specific TEE hardware. It provides the standardized API defined in GP TEE Client API specification. For devices which are not equipped with a real hardware-based TEE, it can allow developers to debug TAs before building them for a real TEE. Figure 2.3 shows Open-TEE architecture.

Its main entry called *opentee-engine* is an executable file which launches two Linux daemons (*Manager* and *Launcher*).

Manager functions as a TEE OS whose responsibilities include dealing with connections to *Launcher* and TAs, monitoring their states and providing resources for TAs. CAs can utilize the pre-agreed socket file to communicate with it using GP TEE Client API which is wrapped into an extra library named *libtee*.

Launcher is responsible for deploying TAs which normally come with the form of an Executable and Linkable Format (ELF) shared library object. Following the *zygote* design pattern, it can create TAs very efficiently by preloading shared libraries. All the TAs are only loaded once before the start of Open-TEE.

2.7 OmniShare TA

OmniShare [29] is an application enables sharing encrypted cloud storage between different authenticated devices [29]. To encrypt the data in cloud storage, it has a root key used to generate directory keys which then encrypt the files under that directory. The root key can be regarded as the master key which must be stored in a place protected by the TEE. So the OmniShare TA is developed to manage the master key. It can generate the root key inside of Open-TEE and corresponding directory keys. It also provides two basic encryption and decryption operations. Currently it uses open authentication for a prototyping purpose.

2.8 SEAndroid

SEAndroid is short for Security Enhanced Android which introduces Mandatory Access Control (MAC) to Android. The Android security mode is enhanced in two places. The first one is in the application permission layer. For each application, the developers have to specify the permissions the application requires which are stored in a manifest file, such as making a phone call or accessing to the Internet. In addition, before the installation of the application, the user must specify what permissions are allowed for this application. Due to the fact that Android uses the Linux kernel as its basic foundations, a process isolation and sandboxing mechanism provided by its kernel can also enhance the Android security mode. Before Android 4.2, its Linux kernel only provides Discretionary Access Control (DAC). Due to the shortcomings of the DAC, SELinux [21] is introduced by the National Security Agency to provide MAC for the Linux. So they decided to apply SELinux to the Linux kernel of Android which leads to the SEAndroid. By adopting the existing Android security mode as mentioned above, they have made the following changes to the Android:

1. Applying the SELinux to the Linux kernel of Android;
2. Offering a set of middleware MAC extensions to the application permission layer.

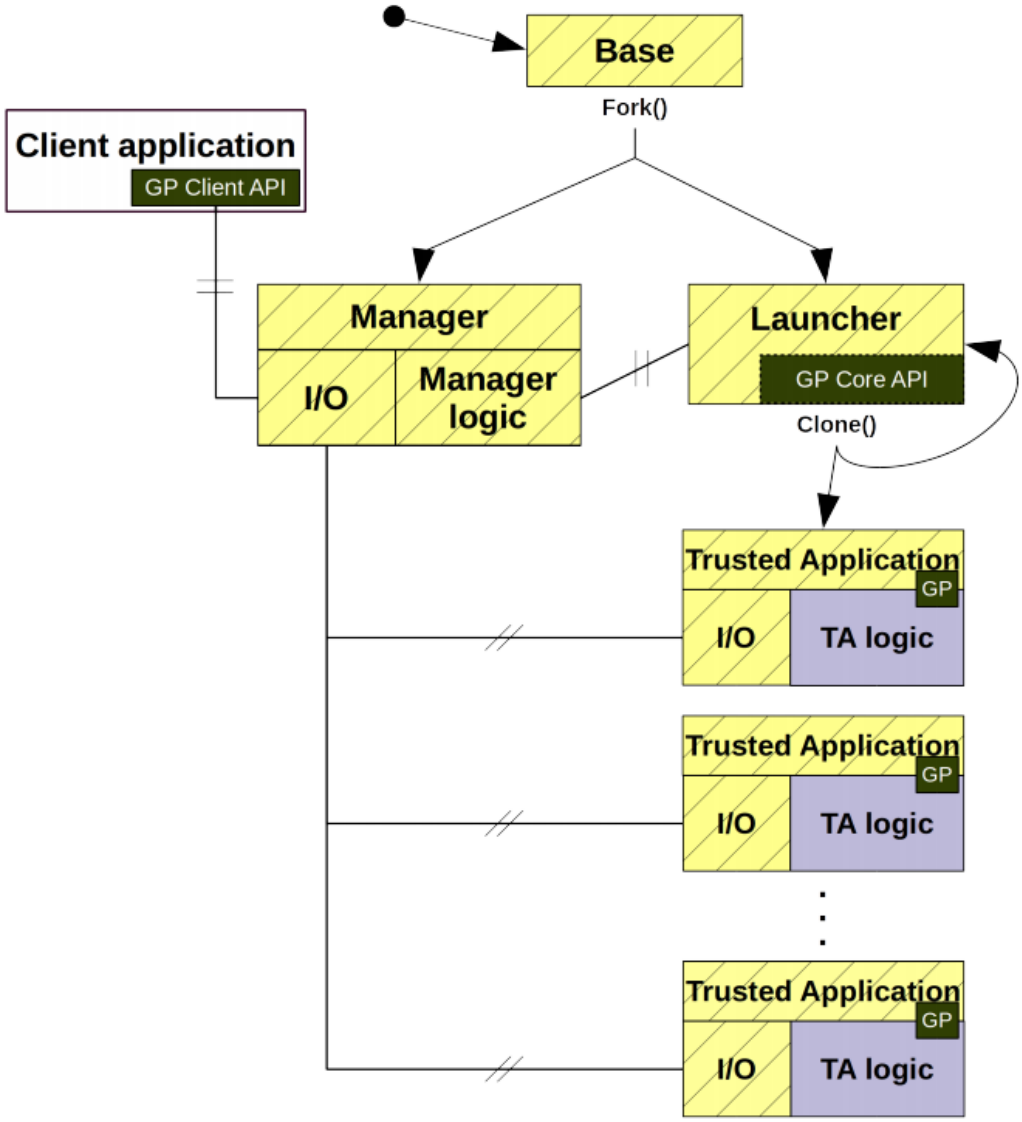

Figure 2.3: Open-TEE Architecture [22]

Chapter 3

Design Overview

This chapter lists all requirements that OT-J should meet in order to evaluate its functionality and usability. In addition, it also presents a system overview of our solution to demonstrate how to make use of OT-J to achieve the original purpose without dealing with native code development.

3.1 Requirements

We identify the following criteria as the requirements for OT-J.

R1: Compliance As a Java wrapper for the GP TEE Client API, OT-J should achieve the same functionality as the GP TEE Client API.

R2: Java Convention OT-J targets the Java developers. So it should conform to Java conventions.

R3: Ease-of-use No need to deal with native code in the development process of CAs.

Another important criteria is performance which is often brought to front when proposing a solution. But it is omitted in here. The reason is that the prototype implementation which is used to evaluate the Java API uses Open-TEE as the GP-compliant TEE. As stated in section 2.6, Open-TEE is a not a real hardware assisted GP-compliant TEE. So the performance can be different when dealing with the GP-compliant TEEs equipped with hardware components.

3.2 System Model

See in figure 3.1, the system model of the proposed solution consists of six components:

Java CA the potion of its code to interact with a TEE/TA is written purely in Java.

OT-J Adaptor is the realization of OT-J. The effort of native code development and transformation between the Java API and C API happen in here. It may consists of many small libraries due to different use cases. An example can be seen in our prototype implementation in Chapter 5. The *OT-J* can be shared among different CAs so that a duplication of efforts can be avoided.

Libtee is a native library which implements the GP TEE Client API. It communicates with a TEE via the GP TEE Core API on behalf of CAs and exposes GP Client C API to upper layer *OT-J*. Then the *OT-J* can interact with it using the Java Native Interface (JNI). The *Libtee* is specific to a particular GP-compliant TEE.

REE OS at the bottom of the left side is an Android OS;

TEE OS on the right side must be GP-compliant which means its implementation must conform to the GP TEE Core API specification[9] so that the Libtee can interact with it.

Native TA contains a set of operations which can be utilized by CAs.

There are different ways to implement this general architecture based on different use cases. For instance, Chapter 5 describes one type of implementations in the context of Android development and Open-TEE. Due to the integration of Android components, its architecture is more complex but still follows this general architecture as proposed in this chapter.

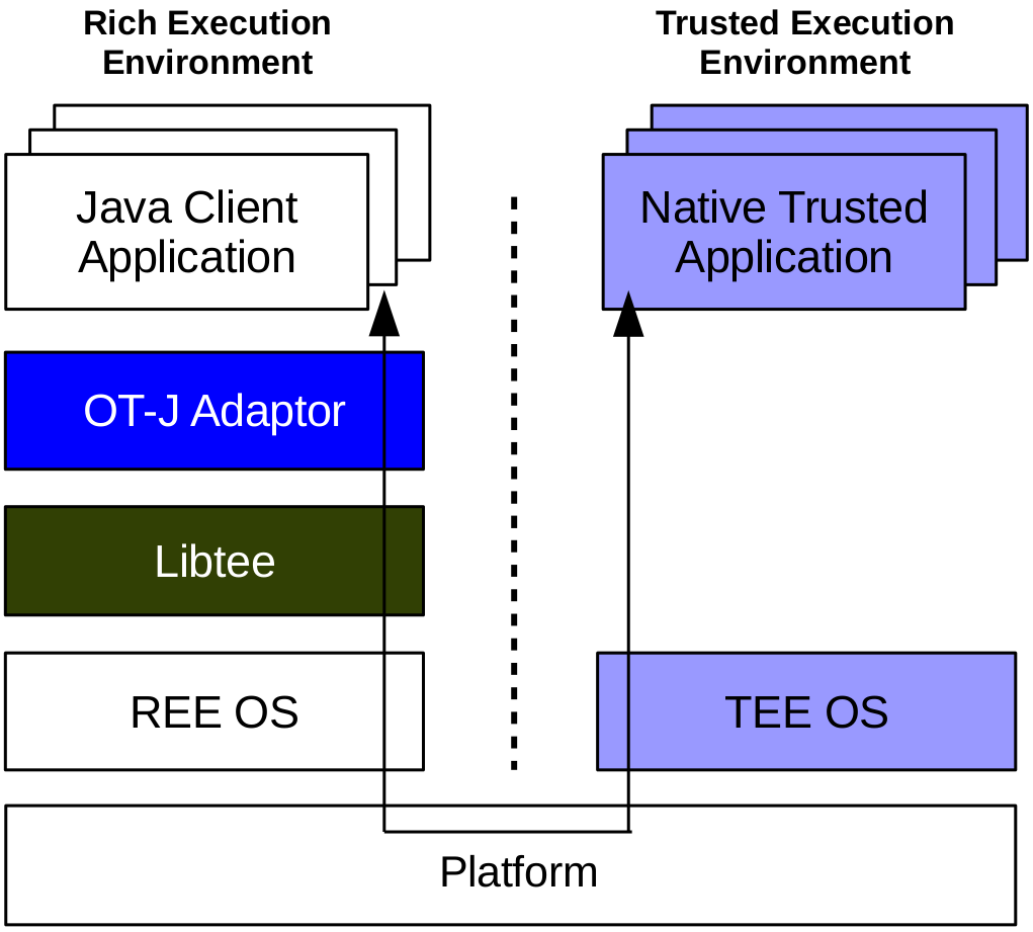

Figure 3.1: System model with Java API *OT-J*

Chapter 4

API Design

This chapter describes the design philosophy behind the OT-J which is based on the GP TEE Client API (see in table 2.1). Some notions, such as *context*, *session* and *shared memory* etc, also originate from it. These entities do have similar meanings in our context. A more detailed explanation can be found in the descriptions for each of them.

In general, a CA wants to utilize functions that a TA provides. To meet its needs, OT-J provides two kinds of functions for the CAs:

1. Communication schemes with a TEE/TA. They are the first and last API for a CA to utilize;

2. Data exchange scheme. After a connection has been successfully established, they can be used to define how the data should be encapsulated and exchanged between a CA and a TEE/TA.

Any abortion or failure of these functions may lead to an inconsistent state between the CA and the TEE/TA. So there is a set of exception classes which are used to indicate such a scenario. As usual, it is the developers' responsibility to handle these exceptions if they occur. The complete OT-J is presented in Appendix A.

4.1 Motivating Example

In order to help explain OT-J, two block of example codes which achieve the same functionality are brought for comparisons. Listing 4.1 example code uses GP TEE Client API and Listing 4.2 utilizes OT-J. The detailed descriptions about the example code are distributed into different sections as followed. For the clarity , these Listings do not show all the parameters of each API.

```
TEEC_Result ret;
TEEC_Context context = {0};
TEEC_SharedMemory shared_memory = {0};
TEEC_Session session = {0};
TEEC_Operation operation = {0};
uint32_t retOrigin;

ret = TEEC_InitializeContext(&context, ...);
if( ret != TEEC_SUCCESS ) return ret;

ret = TEEC_RegisterSharedMemory(&context, &shared_memory, ...);
if( ret != TEEC_SUCCESS ) return ret;

ret = TEEC_OpenSession(&context, &session, ...);
if( ret != TEEC_SUCCESS ) return ret;

operation.params[0].memref.parent = &shared_memory;
operation.params[1].value.a = a;
operation.params[1].value.b = b;

ret = TEEC_InvokeCommand(&session, CMD_DO_ENC, &operation, &
    retOrigin);
if( ret != TEEC_SUCCESS ) return ret;

TEEC_CloseSession(&session);

TEEC_ReleaseSharedMemory(&shared_memory);

TEEC_FinalizeContext(&context);
```

Listing 4.1: GP TEE Client API example code

```
try {
    ITEEClient.IContext context = client.initializeContext(...);

    ITEEClient.ISharedMemory sharedMemory = context.
    registerSharedMemory(buffer, ...);

    ITEEClient.ISession session = context.openSession(...);

    ITEEClient.IValue value = client.newValue(a, b, ...);
    ITEEClient.IRegisteredMemoryReference rmr = client.
    newRegisteredMemoryReference(sharedMemory, ...);

    ITEEClient.IOperation operation = client.newOperation(rmr,
    value);

    session.invokeCommand(CMD_DO_ENC, operation);

```

```
        session.closeSession();
16
        context.releaseSharedMemory(sharedMemory);
18
        context.finalizeContext();
20  }
    catch (TEEClientException e)
22  {
        // handling exception here.
24      retOrigin = e.getReturnOrigin();
    }
```

Listing 4.2: OT-J example code

4.2 Communication Scheme

GP TEE Client API has the notion of a *context* to a TEE and *session*s to individual TAs. They stand for the handle to the connections to a TEE or TAs in the communication scheme. Their relationships are described in figure 4.1. As stated in the beginning of this chapter, the ultimate goal for the communication scheme is to enable the interactions between a CA and TAs running inside of a TEE.

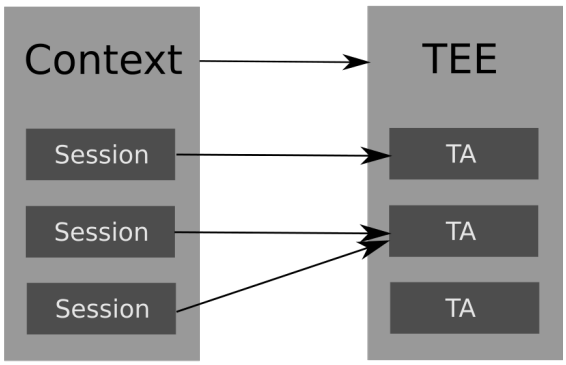

Figure 4.1: Relationship between *context* and *session*

Preparation Before any interaction with a remote TEE/TA happens, the CA must prepare its own environment by calling a factory method *new-*

TEEClient(), which will return an *ITEEClient* interface (see the *client* variable in line no.2 of Listing 4.2). This interface acts as the main entrance to OT-J. More information about it can be found in the OT-J documentation in the appendix. For the GP TEE Client API, developers have to link to a particular Libtee which is specific for the TEE.

4.2.1 Context

The *context* is a general abstraction of a basic communication layer between a CA and a remote TEE. All interactions of a CA with a remote TEE must happen in a valid *context*.

In GP TEE Client API, there is one data structure noted as *TEEC_Context* acting as a handle to an initialized *context* within a TEE. Based on C programming convention, a valid *TEEC_Context* instance must be passed to several C API calls, such as *TEEC_FinalizeContext*, *TEEC_OpenSession* and *TEEC_RegisterSharedMemory*.

In OT-J, the *context* comes as a form of *IContext* interface under *ITEEClient* interface.

4.2.1.1 Initializing a context

To establish a connection to a TEE, a *context* must be initialized within the TEE. In case there are multiple TEEs residing in one mobile device and the Libtee supports each of them, the CA must specify which TEE it wants to connect to by giving the name of TEE. If the name is empty, the *context* will be initialized in a default TEE.

In GP TEE Client API, the *TEEC_InitializeContext* API is used to initialize a new context. Based on the return code (a *TEEC_Result* instance) from the API call, it can be determined whether a valid *TEEC_Context* instance has been initialized or not (see line no.9 of Listing 4.1).

However, it is a different form of use in the OT-J. To initialize a *context*, a method *initializeContext* must be called from the returned *ITEEClient* instance obtained in the preparation process (see line no.2 of Listing 4.2). If this call succeeds, a valid *IContext* interface will be returned as such a valid *context* has been initialized in a remote TEE. If not, a *TEEClientException* will be thrown. For the discussion of the thrown *exception*, please refer to 4.4. An *IContext* interface defines all the API functions in which a valid *context* is needed, including:

1. finalize the context (see 4.2.1.2);
2. open a session (see 4.2.2.1);

3. register shared memory (see 4.3.1.1);
4. release shared memory (see 4.3.1.3).

4.2.1.2 Finalizing a context

If there is not need to interact with the TEE, all shared resources must be released and the context must be finalized. In GP TEE Client API, the *TEEC_FinializeContext* must be called by providing a valid *context* handle initialized previously (see line no.28 of Listing 4.1). In OT-J, the *finalizeContext* function defined in the *IContext* interface must be called (see line no.19 of Listing 4.2).

4.2.1.3 Requesting cancellation

There are two APIs where an *operation* can be involved: *open session* and *invoke command*. The description of the *operation* can be found in section 4.3.4. If one of these two APIs takes a long time for the TEE/TA to process, it is possible for the CA to cancel the function call. We assume that one thread is making one of these two API function calls. This thread will be blocked util the TEE/TA finished and returned. At the same time, when the thread is blocked, another thread can cancel the API function call by making a *request cancellation* call by referencing the same *operation* parameter. In GP TEE Client API, the *TEEC_RequestCancellation* function takes a *TEEC_Operation* as its only parameter. In OT-J, the corresponding API is called *requestCancellation* in a valid *IContext* which also consumes only one *IOperation* instance.

4.2.2 Sessions

There are possible many TAs running inside of a TEE. It is necessary to provide another layer on top of the context to enhance the access control for each connection from the CA to a TA. The *session* is an abstraction of the communication layer between a CA and a TA on top of a valid *context*. So a *session* is valid only in a *context*. All interactions with a TA must happen in a valid *session*.

4.2.2.1 Opening a session

In general, to communicate with a remote TA, a *session* must be opened. Before opening a *session*, the CA must have enough knowledge about the TA that it wants to talk to, such as the Universally unique identifier (UUID)

of the TA. What's more, the CA must provide correct data to authenticate itself to the TA. This can be done by making an agreement on authentication data with the TA beforehand. During opening a *session*, the corresponding authentication data then can be passed to the TA. Of course, these data must be encapsulated using a scheme stated in the next section. Giving any incorrect data will lead to a failure of the open-session operation.

In GP TEE Client API, the *TEEC_Session* structure is used to hold the handle to a *session*. It can be only obtained by making a *TEEC_OpenSession* API call (see line no.14 of Listing 4.1). In order to get a valid *session* handle, a valid *context* handle must be given to the API. In addition, along with the authentication data, a *TEEC_UUID* of the target TA with which the CA needs to interact must also be provided. In later interactions with the TA, which require a valid *session* handle, previously returned *TEEC_Session* must be passed into corresponding API functions, such as *TEEC_CloseSession* and *TEEC_InvokeCommand*.

In OT-J, we use *ISession* as a handle to a *session*. There is no need to pass the *ISession* instance to any API function. Instead, these functions which need the involvements of the *session* are all defined in the *ISession* interface, such as *closeSession* and *invokeCommand*. They are only valid when a valid *ISession* is present. A valid *ISession* can be only obtained by calling the *openSession* function in a valid *IContext* along with correct authentication data (see line no.6 of Listing 4.2).

4.2.2.2 Closing a session

When there is no further interaction with the TA, the corresponding *session* should be closed to release occupied resources. There is one API function named *TEEC_CloseSession* in GP TEE Client API. The only parameter it consumes is a *TEEC_Session* instance (see line no.24 of Listing 4.1). There is no return value to indicate the status of close-session operation. So after this function call, the *TEEC_Session* must not be reused. In OT-J, to close a specific *session*, the function *closeSession* which takes no parameter in the related *ISession* must be called (see line no.15 of Listing 4.2). It also make no guarantee that the *session* can be closed successfully. But the *ISession* must not be reused again too.

4.2.2.3 Invoking a command

Once a *session* is opened to a target TA, to ask the TA to perform a specific operation, the CA can tell TA what to do by invoking a command specified with a command id corresponding to a pre-agreed operation. So before that, a

TA must define a list of commands that a CA can invoke. For each command to invoke, it is possible that there are input or output parameters involved. All the parameters are wrapped in an *operation* which is discussed in section 4.3.4.

In GP TEE Client API, there is one API named *TEEC_InvokeCommand* to perform an invoke-command operation (see line no.21 of Listing 4.1). A valid *TEEC_Context* and *TEEC_Session* must be provided to this API along with the command id and an optional *TEEC_Operation*.

In OT-J, the function *invokeCommand* is defined in the *ISession* interface. As long as the *session* is valid, this function can be called by providing only the command id and an *operation* (see line no.13 of Listing 4.2).

For the return status of this invoke-command operation which is indicated by two values (*TEEC_Result* and return origin code) in GP TEE Client API, please refer back to the section 4.4.

4.2.3 Summary

This chapter describes the communication scheme for a CA to interact with a TEE/TA. Figure 4.2 demonstrates the state of the TEE when each API is called. The following section presents how data can be exchanged between the CA and the TEE/TA.

4.3 Data Exchange Scheme

In the normal computing world, there are two general approaches to transfer data information between two processes: message passing or shared memory. The first one is passing byte stream via the interfaces provided by OSs. This approach is only efficient if the byte stream is small enough since the CPU needs to copy the byte stream back and forth between two processes. Another approach is to share a memory space between two different threads within a single process thus avoiding copying the data. Normal OSs only allow one process access to its own memory space for security concerns. But Android allows two processes to share memory spaces within the same application sandbox using *ashmem*. Under such a circumstance, it is possible to share memory space between two threads/processes but with inevitable computational overheads. As a result, this approach is better than the first one if the data to be shared is quite big due to bigger computation efforts of the first solution. To improve the efficiency of data transmission, both approaches should be taken into consideration and one can choose the approach which fits his/her situations.

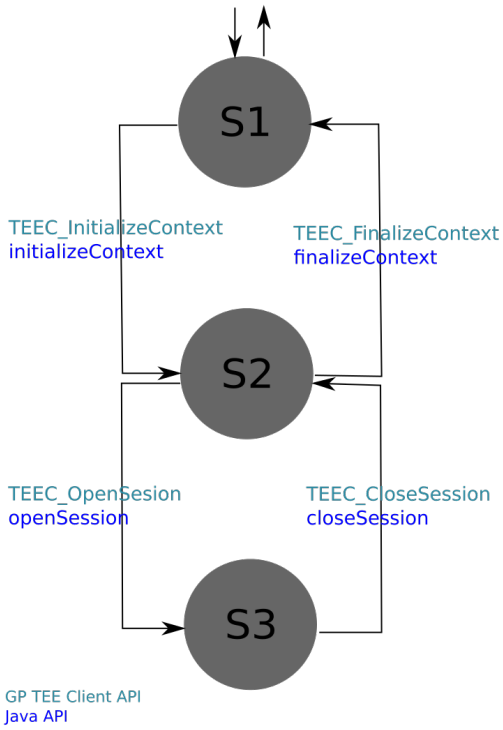

Figure 4.2: Communication Scheme State Machine

These two approaches also apply in the context of TEE since the CA and TA are also processes. So it is crucial to support data transmission between them. The first approach is quite generic in the GP TEE Client API. Different input parameters are copied to/from the TEE/TA when each API is called by the CA. The second approach is mapped to a notion called *shared memory* (see in section 4.3.1).

Once the ways to interact with the TEE/TA are defined, it is necessary to show the data exchange scheme which enables data flow between the CA and the TEE/TA. The data exchange scheme defines what type of data can be exchanged between CAs and TEE/TAs, and how data should be encapsulated. There are *direct* data containers and *indirect* data containers (see in figure 4.3). Direct data containers are the smallest structure which can be referred or encapsulated by indirect data containers. They consists of:

1. *shared memory* is a block of memory in a CA which is shared with a TEE/TA to avoid memory copy back and forth (see section 4.3.1);

2. *value pair* contains two integer values (see section 4.3.2);

3. *UUID* is short for universally unique identifier which is used by CAs to specify which TA it wants to interact with during *open session*. So it must be unique with a TEE (see section 4.3.5).

Excluding the *UUID*, these direct data containers can not be directly transferred between CAs and TEE/TAs. They must be encapsulated in a hierarchy manner, which results in several levels of encapsulations. The motivation of indirect data containers is to provide more flexibility to use the direct data containers and data structure isolations. The outcome of the hierarchy encapsulations are the following indirect data containers:

1. *Registered memory reference* (see section 4.3.1.2);
2. *Parameter* (in section 4.3.3)
3. *Operation* (in section 4.3.4)

It is possible for indirect data containers adding metadata to extend the usage of encapsulated data containers. One example can be found in section 4.3.1.2.

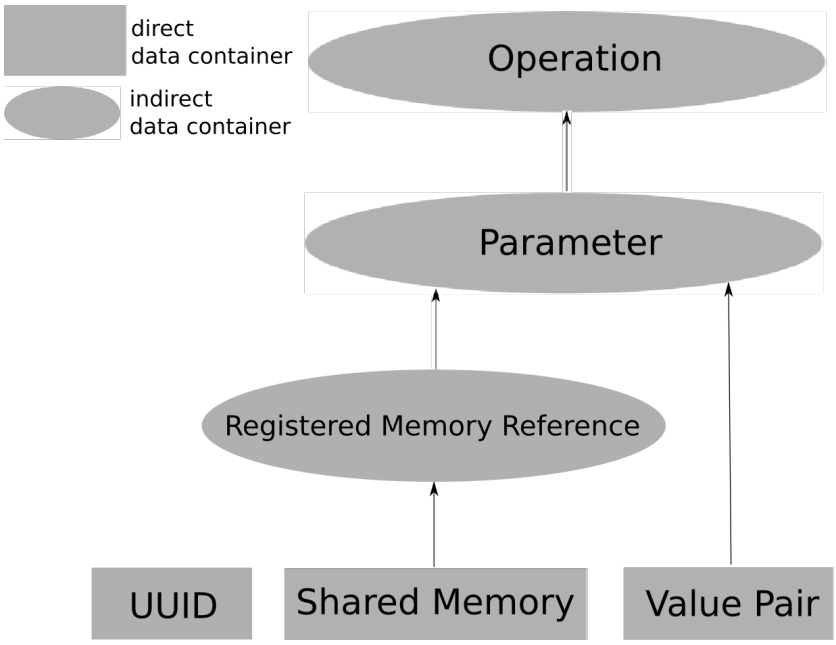

Figure 4.3: Data encapsulation hierarchy

The following two lists are the data types defined in GP TEE Client API and corresponding data interfaces in OT-J. More detailed descriptions about

the transformations from C data types to Java interfaces are discussed in the following sections.

```
interface ISharedMemory {
    int TEEC_MEM_INPUT = 0x00000001;
    int TEEC_MEM_OUTPUT = 0x00000002;
    int getFlags();
    byte[] asByteArray();
}

interface IRegisteredMemoryReference extends IParameter{
    enum Flag{
        TEEC_MEMREF_INPUT(0x0000000D),
        TEEC_MEMREF_OUTPUT(0x0000000E),
        TEEC_MEMREF_INOUT(0x0000000F);

        int id;
        Flag(int id){this.id = id;}
    }
    ITEEClient.ISharedMemory getSharedMemory();
    int getOffset();
    int getReturnSize();
}

interface IValue extends IParameter{
    enum Flag{
        TEEC_VALUE_INPUT(0x00000001),
        TEEC_VALUE_OUTPUT(0x00000002),
        TEEC_VALUE_INOUT(0x00000003);

        int id;
        Flag(int id){this.id = id;}
    }
    int getA();
    int getB();
}

interface IParameter{
    enum Type{
        TEEC_PTYPE_VAL(0x00000001),
        TEEC_PTYPE_RMR(0x00000002);

        int id;
        Type(int id){this.id = id;}
    }
    Type getType();
}

interface IOperation{
```

```
47      boolean isStarted();
}
```

Listing 4.3: Java data interfaces

4.3.1 Shared memory

The notion of *shared memory* in GP TEE Client API specification [7] is a design to enable data transfer between a CA and a TA. Firstly, a block of memory within the CA must be registered as a *shared memory* to a TA. Then the TA can also operate on it. Note that all shared memory are CA driven which means the actual memory space containing the share memory are only located in the CA. The reason is that normally TAs only have limited memory spaces and it is also dangerous to allow CAs to operate on the memory space of TAs from the perspective of security concerns. Since the memory space resides in the CA, the CA can use it freely whether it is registered as a *shared memory* or not. The access control of this *shared memory* for the TA is enhanced by providing indication flags during the action of registering the *shared memory*. After it has been released, the TA will no longer neither read its content nor write to it. The notion *shared memory* is represented as *TEEC_SharedMemory* in GP TEE Client API (see line no.1-7 of Listing 2.1) and *ISharedMemory* in OT-J (see line no.1-6 of Listing 4.3). For the constraints caused by real world implementation environment, please refer to section 5.3.

4.3.1.1 Register shared memory

As stated above, to allow a TA to access the *shared memory*, the CA must register it to the TA. In GP TEE Client API, the *TEEC_RegisterSharedMemory* must be called (see line no.11 of Listing 4.1) by giving a valid *TEEC_Context* and uninitialized *TEEC_SharedMemory*. The related function in OT-J is *registerSharedMemory* defined in the *IContext* interface (see line no.4 of Listing 4.2).

4.3.1.2 Registered memory reference

Once a *shared memory* is registered, in order to refer the *shared memory* to the TA, the CA must encapsulate it into a *registered memory reference* (RMR), which is a container for a *shared memory* and also provides flexibility to use the *shared memory*. The RMR is noted as *TEEC_RegisteredMemoryReference* in GP TEE Client API (see line no.9-14 of Listing 2.1) and *IRegisteredMemoryReference* in OT-J (see line no.8-20 of Listing 4.3). A *TEEC_RegisteredMemoryReference*

can be created by just populating its fields. To create an *IRegisteredMemoryReference*, the CA must call *newRegisteredMemoryReference* which is a function defined in the *ITEEClient* interface (see line no.9 of Listing 4.2).

4.3.1.3 Release shared memory

When a *shared memory* is no longer needed, it should be released so that the TA can no longer access it. This can be achieved with the help of the function *TEEC_ReleaseSharedMemory* in GP TEE Client API (see line no.26 of Listing 4.1) and *releaseSharedMemory* of the *IContext* interface (see line no.17 of Listing 4.2).

4.3.2 Value pair

Value pair consists of two integers, which is used to transfer simple data without the need to register a shared memory. See in figure 4.3. The *value pair* must be encapsulated into a *parameter* to enable its transmissions between CAs and TAs. It is defined as *TEEC_Value* in GP TEE Client API (see line no.16-20 of Listing 2.1) and *IValue* in OT-J (see line no.22-33 of Listing 4.3). A *TEEC_Value* instance can be created by filling its fields. For *IValue*, the function *newValue* from the *ITEEClient* interface must be called.

4.3.3 Parameter

Parameter is the smallest unit for an *operation* (see in next section). In GP TEE Client API, *parameter* is represented as a C union called *TEEC_Parameter* (see line no.22-27 of Listing 2.1). It can be either a *TEEC_TempMemoryReference*, *TEEC_RegisteredMemoryReference* and *TEEC_Value*. In OT-J, the *parameter* is an interface called *IParameter* (see line no.35-44 of Listing 4.3). Since there is the notion of union in Java, we have defined the *IParameter* as super interface for both *IRegisteredMemoryReference* and *IValue*. There is no corresponding Java data type for *TEEC_TempMemoryReference*. For the reason why it is omitted, please see section 4.5.1.

4.3.4 Operation

Operation is the only indirect data container which can be directly transmitted between CAs and TAs. The *operation* can take up to 4 *parameter*s. The related data structure in GP TEE Client API is *TEEC_Operation* (see line no.29-35 of Listing 2.1). Excluding the *TEEC_Parameter*s it includes, there are two metadata fields. The first one is *started* which is used to indicate

the current status of the *TEEC_Operation*. Another field is *paramTypes*. It contains the type of the included *TEEC_Parameter*s. Once it is passed into an either *open session* or *invoke command* function call, it can be used to cancel these two function calls which is either started or in a pending state within the TEE/TA.

In OT-J, the data type mapped with the *operation* is *IOperation* (see line no.46-48 of Listing 4.3). The *started* field is transformed to a *isStarted* function. The *paramTypes* field is not needed since it can be implied by the *IParameter*s it includes.

To create a *TEEC_Operation* instance, the CA just needs to populate its fields. For our *IOperation* interface, there are 5 overloaded functions with the name of *newOperation* from *ITEEClient* interface, which take 0 to 4 *parameters*. The function in line no.11 of Listing 4.2 is one overloaded function which takes two *IParameter*s.

4.3.5 UUID

UUID is the only direct data container which can be directly transmitted between CAs and TAs. For the CAs to distinguish the TAs, the UUID of each TA must be unique within one TEE. It is only needed during *open session* to specify which TA the CA wants to connect to. UUID is short for Universally Unique IDentifier which is defined in RFC4122 [17]. In OT-J, we take the existing UUID data structure from Java package java.util which also conforms to the RFC4122 standard.

4.4 Exceptions

The return status of each API function call is indicated in a different way between C and Java. With the C binding in GP TEE Client API, for those functions with a return value, the return status is indicated using global constant values (an unsigned 32-bits integer). For instance, *TEEC_SUCCESS* is returned if one API call succeeded and there is a return value for it (see example code in line no.9, 12, 15 and 22 of Listing 4.1). An error code is returned if failed. There are many error codes defined in the GP TEE Client API specification. Each error code corresponds to one kind of error.

However, this is implemented differently in OT-J. If one function call succeeded, a valid interface is returned (see example code in line no.2, 3 and 4 of Listing 4.2). There is no corresponding data structure to map *TEEC_SUCCESS*. The type of returned interface is different in different function calls. If failed, an exception will be thrown (see line no.21 of Listing 4.2).

TEEClientException is the superclass for all exceptions that we have defined. Each exception extends the *TEEClientException* and is mapped back to one error code. So each error code defined in GP TEE Client API specification has an exception corresponded to it. In our prototype implementation, we also re-threw the exceptions coming from Android to our own exceptions.

For *TEEC_OpenSession* and *TEEC_InvokeCommand* functions in GP TEE Client API, there is another field called *return origin* (an unsigned 32-bit integer) used to indicate the original location of the error (see line no.24 of Listing 4.1). Currently, there are four global constant values for this field:

1. *TEEC_ORIGIN_API*;
2. *TEEC_ORIGIN_COMMS*;
3. *TEEC_ORIGIN_TEE*;
4. *TEEC_ORIGIN_TRUSTED_APP*.

In OT-J, these four values are transformed into an enum named *ReturnOriginCode*. When an exception is thrown from these two functions above, the CA can know the origin of the error by calling the function called *getReturnOrigin* in the thrown exception (see line no.16 of Listing 4.2).

4.5 Omitted Data Types and Functions

Due to the different environments in which the GP TEE Client API and OT-J operate, the following two concepts have been omitted in OT-J.

4.5.1 TEEC_TempMemoryReference

In GP TEE Client API, *TEEC_TempMemoryReference* is another way of sharing a block of memory within a CA to a TA, which is similar with *TEEC_SharedMemory*, both with a buffer, size field and corresponding flags to indicate the I/O direction. However, unlike *TEEC_SharedMemory*, the *TEEC_TempMemoryReference* is only valid for one time *operation* which means it cannot be reused by TAs in a sequence of *operations*. In other words, the same *TEEC_TempMemoryReference* will be regarded as different instances in TAs. The major benefit of the *TEEC_TempMemoryReference* is that it can avoid two operations which are mandatory for *TEEC_SharedMemory*: *TEEC_RegisterSharedMemory* and *TEEC_ReleaseSharedMemory*. In addition, *TEEC_TempMemoryReference* can be used directly in the *TEEC_Parameter*.

But the *TEEC_SharedMemory* needs to be encapsulated into a *TEEC_RegisteredMemoryReference* which, then, can be used in the *TEEC_Parameter*. The biggest drawback of the *TEEC_TempMemoryReference* is non reusable. This can be problematic when the same *TEEC_TempMemoryReference* is used for many times. For a TEE to map a memory space that a *TEEC_TempMemoryReference* referred from the CA to a TA, (how the TEE maps the memory space corresponds to real world implementations), there is computation cost for this operation. The computation cost will become considerably larger when the same *TEEC_TempMemoryReference* is used frequently. Since the *TEEC_SharedMemory* is reusable, there is no such an issue for it. So for the simplicity of OT-J, we have decided to remove the *TEEC_TempMemoryReference* and recommended using the *TEEC_SharedMemory* instead.

4.5.2 TEEC_AllocateSharedMemory

The *TEEC_AllocateSharedMemory* will allocate a new memory space as a *shared memory* but the *TEEC_RegisterSharedMemory* registers an existing memory space as a *shared memory* instead. One *TEEC_AllocateSharedMemory* API call can finish the following two tasks in a line:

1. allocating a block of memory in the CA;
2. making a *TEEC_RegisterSharedMemory* API call.

This can be helpful since this API can allocate the memory space for the CA without the CA directly to do so. The allocated memory can be explicitly freed during the *TEEC_ReleaseSharedMemory*. This API can free C developers from dealing with memory allocations. But it is different in Java. The deallocation of memory space is controlled by Java Virtual Machine (JVM) which adopts Java garbage collection mechanism. Even there is still one reference to the allocated *shared memory*, it will not be freed by the JVM. This can be troublesome. Consider the case that during the usage of the *shared memory*, it has been passed to another Java object. After the CA called the *TEEC_ReleaseSharedMemory*, that Java object still hold the reference to the *shared memory*. So it will not be freed by JVM. However the *TEEC_ReleaseSharedMemory* is dedicated to do so. This is not a problem in C because C developers can explicitly free the memory space by system provided functions, such as *free*. Since the functionality of the *TEEC_AllocateSharedMemory* can be achieved using the *TEEC_RegisterSharedMemory*, along with the concerns in Java, we have decided not to map *TEEC_AllocateSharedMemory* C API to OT-J.

4.6 Summary

The UML diagram of our complete OT-J is presented in figure 4.4. All the data types and API functions are mapped into OT-J. For those omitted, they are explained in section 4.5. To minimize mutability, all the data types in OT-J are represented as public interfaces. All interactions with a data type are predefined in the interface. This hides implementation details, which offer the flexibility for the underlying core library development. In addition, possible exceptions which will be thrown by each function are well documented.

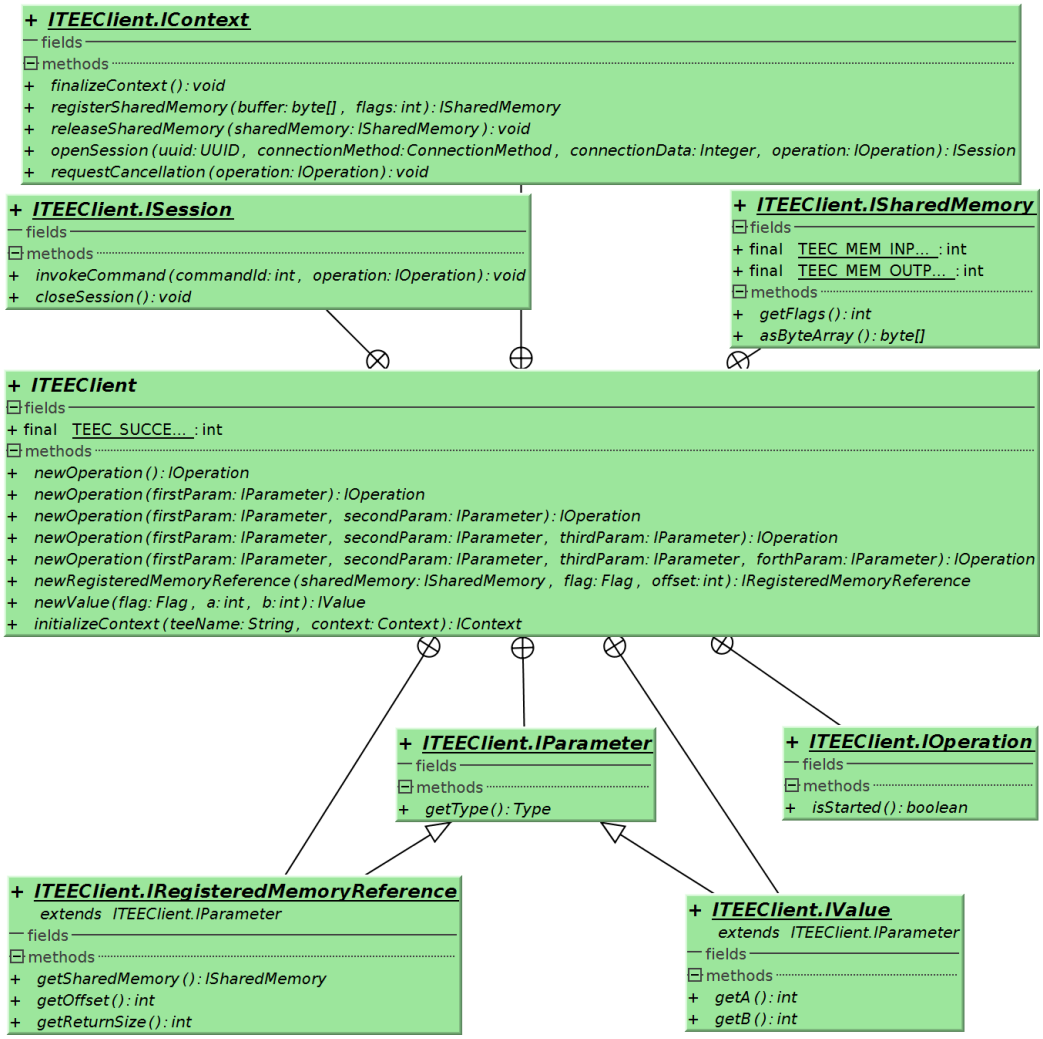

Figure 4.4: OT-J UML Diagram

Chapter 5

Prototype Implementation

To prove the usability of our Java API, we have created a prototype implementation with the Java API fully realized. This chapter describes the principles and design decisions behind implementing the key concepts, such as *shared memory*, based on the Android architecture. Due to constraints of Andriod, the real implementations of these concepts have been leveraged to achieve the maximum functionality and compatibility.

5.1 Overview

As stated previously, Open-TEE is quite useful in the early stage of developing CAs and TAs. Developers can use it for debugging purposes before deploying the TAs into a real TEE. The reason why we choose Open-TEE is that it is GP-complaint TEE solution which is compatible with the GP TEE Client API so with our Java API. Using already existing GP-compliant TEE components can allow us put more focus on our own Java API implementations. The figure 5.1 is the overall architecture for the prototype implementation.

In general, we deploy Open-TEE within an Android application and expose its functionality as an Android service which is named as *TEE proxy service*. This service is responsible for

1. deploying Open-TEE and TAs for only once;
2. communicating with Open-TEE using provided *NativeLibtee* library;
3. handling service requests from CAs and providing resource isolations between these CAs.

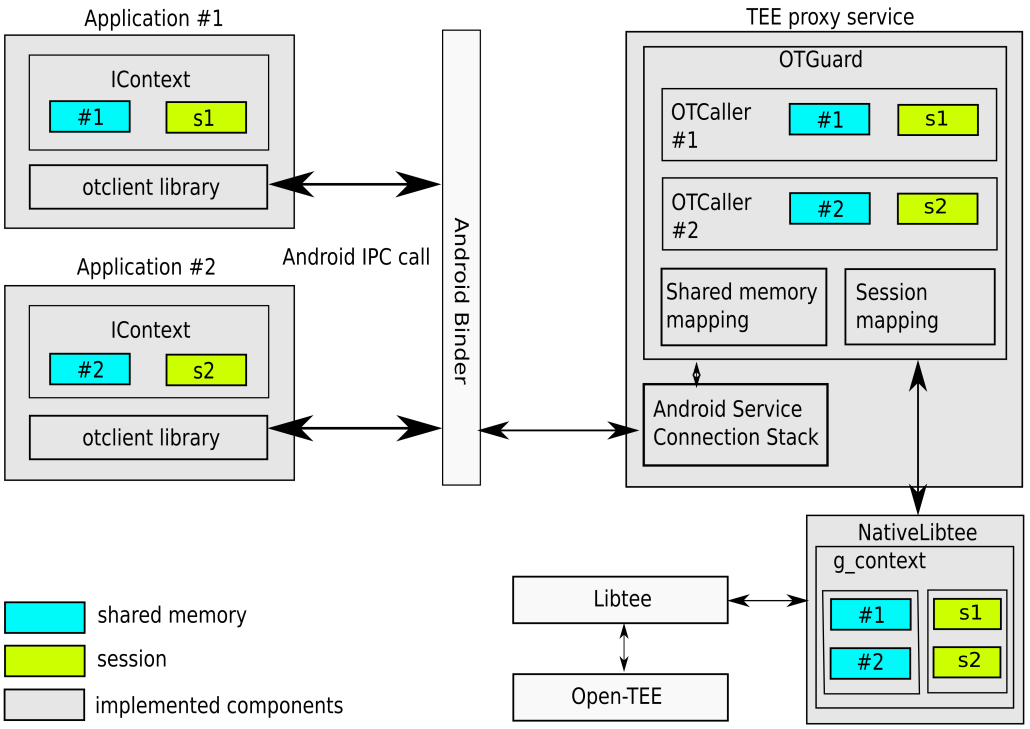

Figure 5.1: Prototype Implementation Architecture

As stated in the Chapter 2, the main entrance of Open-TEE is *opentee-engine* which is the output after building Open-TEE against the Android source tree. It can be directly run as an Android process. If it is launched within one application, other processes running within the same application can use the *Libtee* to communicate with it by using a pre-agreed socket file. But other applications cannot communicate with it due to the SEAndroid [28] restrictions which do not allow socket communication between different applications. In order to comply with the SEAndroid restrictions, we decided to deploy Open-TEE within an application which uses *Libtee* to communicate with it. By adopting this application to an Android service, other applications are also able to interact with Open-TEE via Android Binder.

The work flow of the implementation is:

the *TEEC proxy service* deploys Open-TEE and TAs running in a user mode;

the *TEEC proxy service* exposes its service to CAs using self-defined AIDL interfaces;

CAs connect to the *TEEC proxy service* using provided *otclient* library.

Recall our solution hierarchy presented in figure 3.1. This prototype implementation architecture can be categorized into the following components:

Java CA : Application #1 and #2 which are developed purely in Java and they both want to interact with Native TAs;

OT-J Adaptor : *otclient library* which implements the Java API and communicates with the remote TEE service using Android Inter-process Communication (IPC) call, *TEE proxy service* that exposes its service to Application #1 and #2 together with resource isolations, and *NativeLibtee* implementing the Java call from TEE proxy service in C;

Libtee : *Libtee* which came with Open-TEE and communicates with Open-TEE via GP Core API;

REE OS : Android OS;

TEE OS : Open-TEE a GP-compliant TEE;

Native TAs : TAs running in Open-TEE, which are developed in native code.

The following sections describe more detailed implementations of key concepts issued in both the GP TEE Client API and Java API.

5.2 Java API Implementation

The *otclient* is a full implementation of our Java API. It is responsible for monitoring the mapped resources with the remote *TEE proxy service*. For each Java API function, there are one or two Inter-process communication (IPC) call(s) related with it. Two IPC calls are needed for some Java APIs which can allow the *TEE proxy service* to notify the CA (an example can be found in section 5.3). Benefiting from the synchronous feature of IPC call, implemented Java API is also synchronous which is conformed to GP TEE Client API specification. The figure 5.2 is the calling sequence behind one Java API call.

1. step 1: CA makes a function call *f1* defined in our Java API, such as *initializeContext*;

2. step 2: *otclient* issues an IPC call *f2* related to *f1*;
3. step 3: the corresponding GP TEE Client API function mapped to *f2* is called;
4. step 4: *Libtee* internal call to Open-TEE;
5. step 5: function call returns.

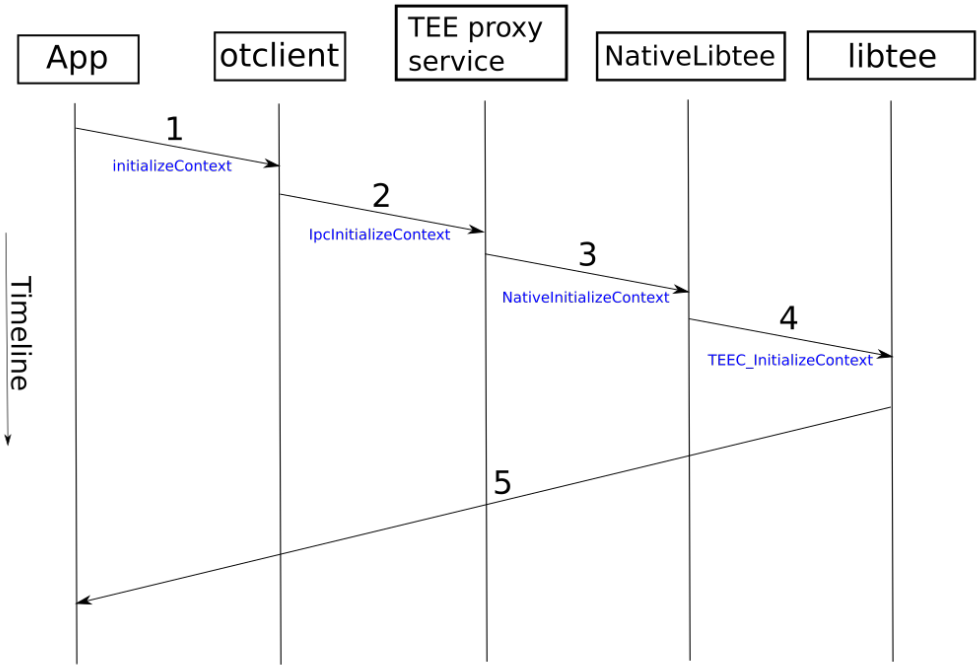

Figure 5.2: API synchronization

Note that all exceptions that the CA can receive are thrown by the *otclient*, which are determined by the error code and possible *return origin code* given by each IPC call. Since the *otclient* is making Android IPC call, it can get a standard *android.os.RemoteException* which is re-thrown by the *otclient* as a *CommunicationException*).

5.3 Shared Memory Implementation

As stated in section 4.3.1, the *shared memory* is a block of memory within a CA which can be shared with a TA. In our implementation, the CA is

an Android application. It is challenging to share a memory block between different Android applications due to the sandbox mechanism of Android. One possible solution is using a kernel driver called *ashmem* but it only allow share memory blocks within one application among different processes. Thus we turn to the second solution. Instead of directly mapping memory block from a CA to a TA, for one *shared memory*, we have two memory blocks corresponding to it. One is in the CA and another one in the *TEE proxy service*. To fulfill the requirements issued in GP TEE Client API specification for shared memory, we only need to synchronize these two memory blocks before and after each API call.

5.3.1 Memory Synchronization

The memory synchronization is achieved by copying the content of the shared memory back and forth between a CA and the proxy service. For how to transfer *shared memory* via Binder driver, please see section 5.4. Since it is expensive to transfer data in the IPC call, our implementation only copies the content of the shared memory under the following two circumstances:

1. copy the shared memory from the CA to TA if its flag is input for the TA;

2. copy the shared memory from the TA to CA if its flag is output for the TA;

5.4 Data Serialization

Without direct mapping of memory blocks, all the data must be passed through Android Binder driver. According to [13], not all the data types can be directly transmitted via Binder especially those self-defined classes. There are two methods to solve this problem. Both of them are used in our implementation. The first one is implementing *Parcelable* interface for each class as stated in [13]. Moreover, an extra AIDL file related to each class also have to be added. This method is used in transmitting the *shared memory* named *OTSharedMemory* which implements the *ISharedMemory* interface.

There is another approach to transmit self-defined classes between a CA and the proxy service by introducing the *protocol buffers* [14]. Since the byte array can be directly transferred via IPC call, using the built-in serialization functions of *protocol buffers* can help us serialize self-defined classes into a byte array. After the byte array is transmitted, these classes can be recreated also using the built-in parsing functions of the *protocol buffers*. One challenge

with this approach is that the size of the serialized data can change after the classes it holds have been modified. If the size increases, it is impossible for the Binder to sync it back to the other side using the original byte array reference if the byte array is specified to do so. To solve this issue, we uses an extra IPC for synchronizing the bigger byte array back, which explains why some APIs are mapped with two IPC calls as stated above.

This is how it works in our implementations. A caller transfers the byte array to a callee along with a callback function which is the second IPC call. Then the callee modifies the byte array which leads to a lager size of byte array compared with the original one. The callee can sync the larger byte array to the caller using the callback function provided previously by the caller. The *operation* named *OTOperation* in our context which implements the *IOperation* interface is transmitted using this method. The main contribution of this approach is that it eases the data manipulations between C and Java. An example is that one byte array can be parsed into correct C structs and Java classes. Consider the case when a CA serializes an *OTOperation* and transmits it to the proxy service. The proxy service then pass it from Java layer to Java Native Interface (JNI) layer which is written in C. The *OTOperation* in a byte array then can be translated into C type of *operation*. Without the help of the *protocol buffers*, we have to either implement our own serialization solutions just like *protocol buffers* which is a duplication of efforts, or access different fields of the *OTOperation* to create an *TEEC_Operation* instance which is hard to deal with and error-prone due to the JNI development conventions.

5.5 Multiplexing

The *TEE proxy service* can allow multiple CAs to connect to it. The requests coming from CAs will end up in the same Open-TEE. So a multiplexing solution must be given between the proxy service and Open-TEE to guarantee resource isolations coming from different CAs. The CAs can be distinguished by the proxy service using their Process Identifier (PID).

There are two solutions to this issue. The first approach which we have taken is presented in figure 5.1. *Application #1* has a shared memory noted as *#1* and a session *s1*. Another application *Application #2* also has a shared memory *#2* and a session *s2*. Within the *TEE proxy service*, an *OTGuard* is responsible for the resource isolations. For each CA, there is an *OTCaller* in the *OTGuard* related to it. All the resources of one CA which are shared with the TEE/TAs are wrapped in one *OTCaller*. Although all shared memory (*#1* and *#2*) and sessions (*s1* and *s2*) are terminated

within the same *NativeLibtee*, benefiting from the upper layer isolations, one CA still not be able to access the resources belonging to another CA. Since this isolation is achieved by the software implementation, we regard it as a weak isolation which is not ideal.

Another approach which can provide a stronger isolation is moving the duty of resource isolations from the *TEE proxy service* to Open-TEE. Although all resources also ended up within the same *NativeLibtee*, by opening one *libtee* for each CA, resources will be mapped into corresponding context within Open-TEE. It is inevitable for the *NativeLibtee* to deal with resource indexing just like what the *OTGuard* does. This approach is better than the first one because of using different contexts for different CAs. In the first approach, there is only one context for all resources from different CAs. So Open-TEE regards them as resources coming from one CA. However, this approach guarantees that one CA can have its own context in Open-TEE. Resources from different CAs can be isolated by Open-TEE.

During the real world implementation, both approaches are applicable. Since the main focus of this thesis is about developing Java API, the prototype implementation just need to prove the usability of the API. So it does not matter which approach is implemented in here as long as it can achieve the goal of resource isolations.

Chapter 6

Evaluation

Based on our prototype implementation, we hereby evaluate the usability of the *OT-J* according to the requirements defined in section 3.1. For the development environment, please see in table 6.1.

6.1 R1: Compliance

The first requirement is that the *OT-J* must provide full coverage of the functionality defined in the GP TEE Client API. After testing each Java API, it will make sense to combine them to achieve specific goals. As mentioned in Section 2.7, OmniShare is an application to share encrypted cloud storage between authenticated devices. It holds a master key to decrypt the encrypted cloud storage, which is currently stored using the Android KeyChain system. Since a GP-compliant TEE can provide integrity protection and secure storage services, the OmniShare TA was developed to provide this feature for the master key. It uses public authentication and has several predefined commands that a CA can invoke, for instance generating a private key, encrypting and decrypting data. So we have one Android test application called *OmniShare tester* (see in figure 6.1) which utilizes the *OT-J* and is developed purely in Java as the CA to use the functionality that OmniShare TA provides. It can be regarded as either Application #1 or #2

Debug device	Nexus 6 running Android 5.1.1
Development tool	Android Studio 2.0
TEE	Open-TEE
TA	OmniShare TA

Table 6.1: Development Environment

in figure 5.1. The OmniShare tester has the following six buttons attached with six corresponding operations and multiple Java API calls are involved in each operation:

Create Root Key button asks the OmniShare TA to generate a root key for later usages (see the log at line no.1 in figure 6.2). This operation comprises of the following API calls in a sequence:

1. *initializeContext*: initializing a context within Open-TEE;
2. *openSession*: opening a session to the OmniShare TA;
3. *registerSharedMemory*: registering a shared memory with a I/O direction as **output** for the TA. The shared memory will be used as a buffer for the generated root key;
4. *invokeCommand*: asking the OmniShare TA to invoke a command with specific command id and parameters (generating root key in here);
5. *releaseSharedMemory*: releasing the shared memory which previously registered;
6. *closeSession*: closing the session to the OmniShare TA;
7. *finalizeContext*: finalizing the connect to Open-TEE.

After a root key is generated, the *Initialize* button will be enabled.

Initialize button just pass the previously generated root key to the OmniShare TA, which is needed in later operations (see the log at line no.2 in figure 6.2). Since we have already released the shared memory used to hold the root key in previous operation (the share memory also is output for the TA), so we need to pass the root key to the TA specifically. This initializing operation is a serial of the following API calls:

1. *initializeContext*: initializing a new context within Open-TEE;
2. *registerSharedMemory*: registering a shared memory with a I/O direction as **input** for the TA. The shared memory will be used as an input buffer to pass the root key to the TA;
3. *openSession*: opening a session to the OmniShare TA with the public authentication method;

4. *releaseSharedMemory*: releasing the shared memory which was used to pass the root key.

Since the session is not closed, the connection between the CA and the TA is still valid. After the initialization operation done, the *Finalize* and *Create Directory Key* buttons are enabled.

Finalize button closes the session previously opened and finalizes the context initialized earlier (see the log at line no.6 in figure 6.2). It consists of the following sequence of API calls:

1. *closeSession*: closing the session to the OmniShare TA;
2. *finalizeContext*: finalizing the connect to Open-TEE.

After this operation, the tester application will not longer be able to interact with Open-TEE (so as the TA) without initializing a new context.

Create Directory Key button requests the TA to generate the directory key for current directory using the root key given previously (see the log at line no.3 in figure 6.2). This operation contains the following API calls:

1. *registerSharedMemory*: registering a shared memory with a I/O direction as **output** for the TA. The shared memory will be used as an output buffer to pass the generated directory keys to the CA;
2. *invokeCommand*: asking the OmniShare TA to create directory keys;
3. *releaseSharedMemory*: releasing the shared memory which was used to retrieve the directory keys.

Now, both the CA and TA have the generated directory keys.

Encrypt Data button requests the TA to encrypt a data buffer (shown in the log noted as *Initial data buffer* in 6.2) using the directory keys (see the log at line no.4 in figure 6.2). The following API calls are wrapped in this operation:

1. *registerSharedMemory*: registering a shared memory with a I/O direction as **input** for the TA. The shared memory will be used as an input buffer to pass the data, which is needed to be encrypted using directory keys, to the TA;

2. *registerSharedMemory*: registering a shared memory with a I/O direction as **input** for the TA. The shared memory will be used as an input buffer to pass the previously generated directory keys to the TA;

3. *registerSharedMemory*: registering a shared memory with a I/O direction as **output** for the TA. The shared memory will be used as an output buffer to pass out the encrypted data;

4. *invokeCommand*: asking the OmniShare TA encrypt the input data using the directory keys;

5. *releaseSharedMemory*: releasing the shared memory which was used to pass out the encrypted data.

6. *releaseSharedMemory*: releasing the shared memory which was used to pass in the directory keys.

7. *releaseSharedMemory*: releasing the shared memory which was used to pass in the unencrypted data.

This operation enables the *Decrypt Data* button.

Decrypt Data button requires the TA to decrypt the encrypted data buffer using the directory keys. Its operation has a highly similar calling sequence to the *Encrypt Data* operation (see the log at line no.5 in figure 6.2). So it is omitted in here. The outcome of this operation leads to the same content as the *Initial data buffer* thus proving the correctness of our implementations.

After all the buttons are clicked in a certain order, we can see the output on the screen in figure 6.2.

As stated in section 4.5.2, the *TEEC_AllocateSharedMemory* can be replaced by the *TEEC_RegisterSharedMemory* thus its functionality fulfilled by the *registerSharedMemory* Java API in our case. We still have the *RequestCancellation* to test. Normally, this API is used to cancel an operation including either the *openSession* or *invokeCommand* which takes a considerable long time to proceed. However, it is very difficult to create a such scenario to test this API. Without changing the source code of the OmniShare TA, we cannot hang the TA for a certain among of time while waiting for the *requestCancellation* to happen. But we have proved that the *requestCancellation* is observed in the NativeLibtee. Other Java APIs are used in the OmniShare tester application and proven to function very well.

In summary, our Java API does provide the same functionality as the GP TEE Client API.

6.2 R2: Java Convention

To design an API which is familiar to Java developers, we have focused on making it easy to use, hiding unnecessary information and defining classes as immutable as possible. The outcome is that all the C structs have been transformed into Java classes (see in figure 6.3). C APIs are distributed into these Java classes based on their scopes (see in chapter 4).

We also switched the return value based error handling method to the exception based error handling mechanism which is more common in Java. Last but not least, the Java factory method design pattern is normally used to create an object without specifying what type of class of the object going to be created. It is introduced into our Java API. For instance, the superclass *IParameter* has a method called *getType*. Both *IRegisteredMemoryReference* and *IValue* subclass the *IParameter* and implement the *getType* method. So once the *getType* of one *IParameter* instance is called, it will return a value based on the child of this *IParameter* instance.

6.3 R3: Ease-of-use

The first element of ease-of-use is that the Java developers no longer need to write similar native code. They just simply import the OT-J into their project and then start using the provided Java API. Moreover, we have two OmniShare tester applications which have the same functionality to show how much line of code can be reduced by using the OT-J. One (*OmniShare tester*) is presented in previous section which is written in pure Java and utilizes the OT-J. Another one (*OmniShareJni tester*) is a combination of Java and native code which does not use the OT-J. For the codes which are responsible for communicating with Open-TEE, omitting the major comments, the *OmniShareJni tester* has proximately 661 lines of code while the *OmniShare tester* only have around 368.

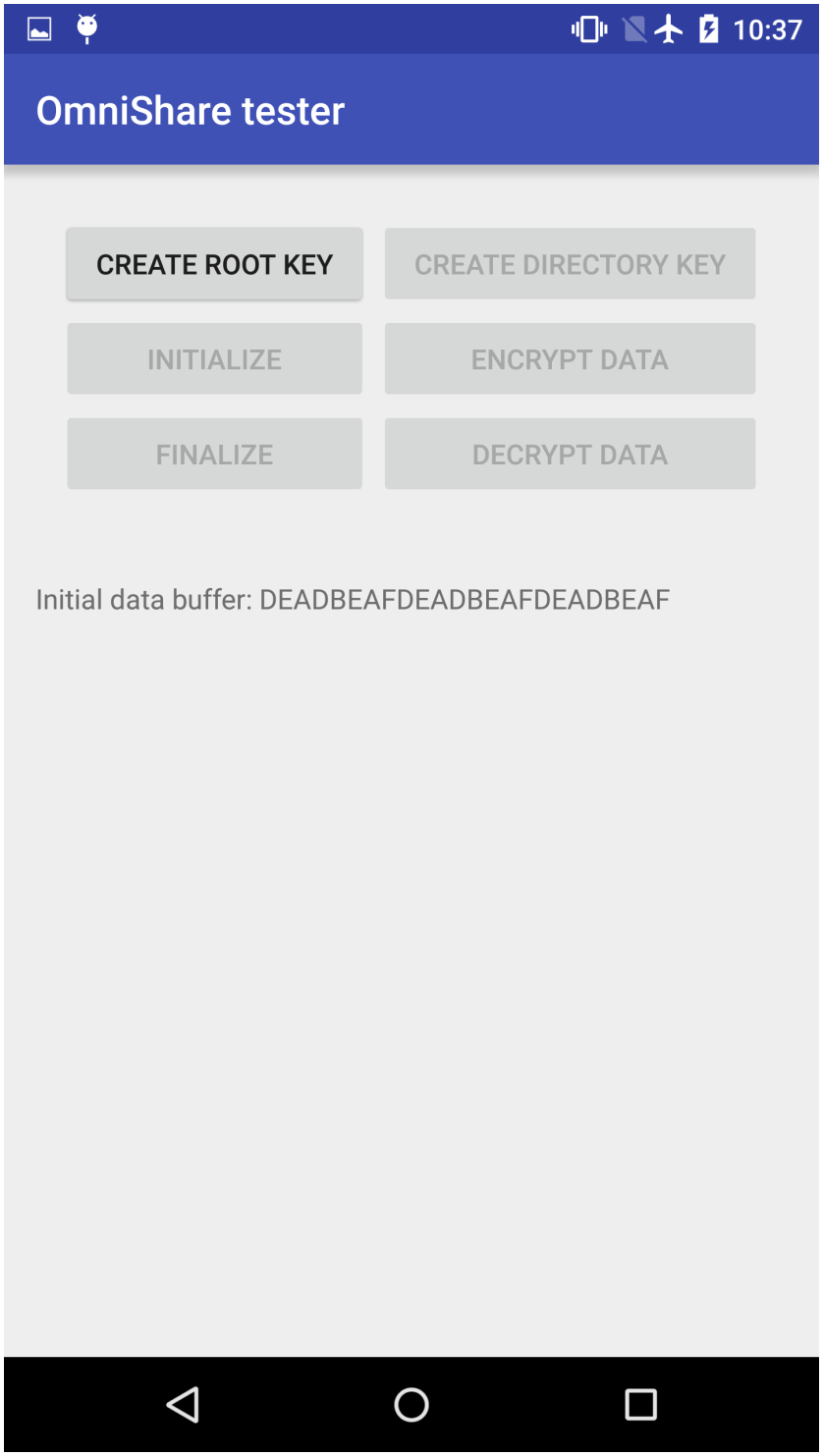

Figure 6.1: OmniShare tester application initial state.

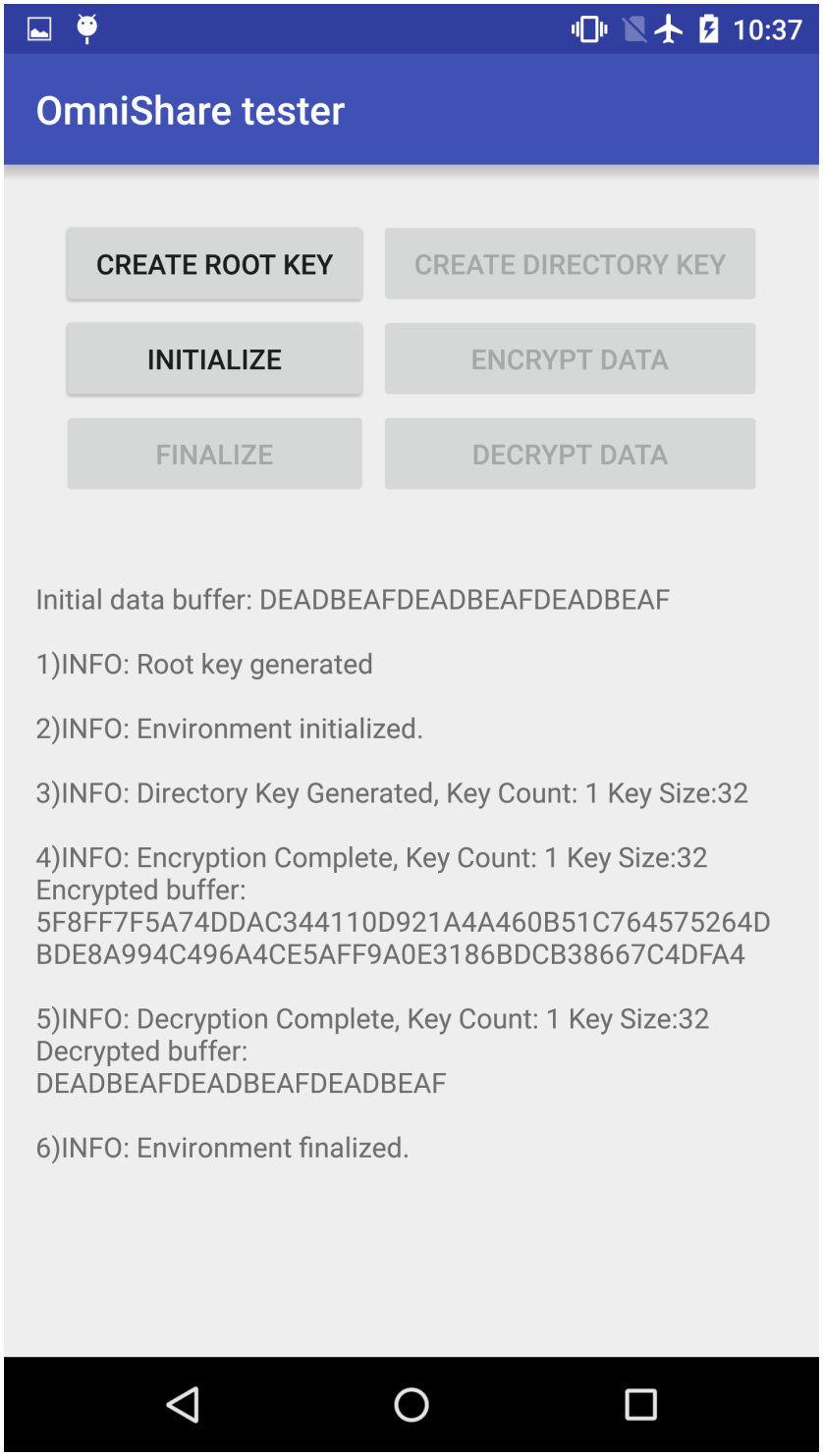

Figure 6.2: Log messages printed on OmniShare tester application after all buttons are pressed in a sequence.

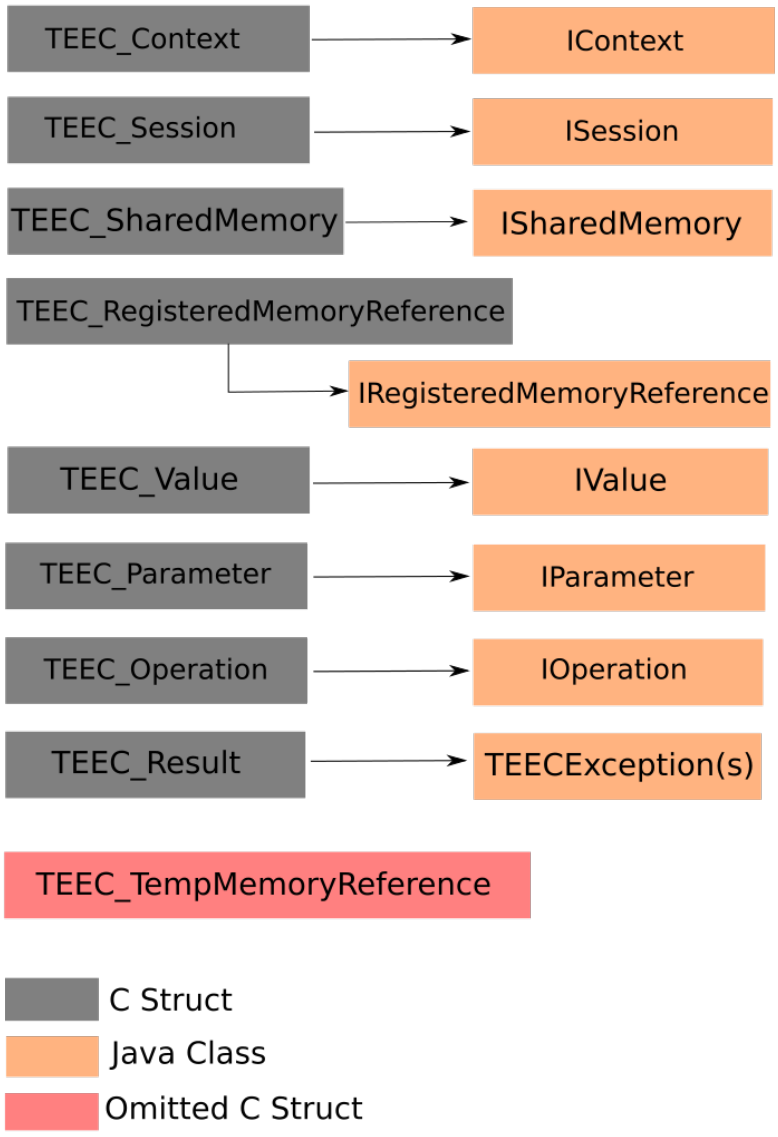

Figure 6.3: C structs to Java Classes

Chapter 7

Related Work

As far as we are aware, to date there is *no* open project which maps the GP TEE Client API to the Java world. But several projects have devoted their efforts to all the usage of secure hardware for applications written in native code to Java. By focusing on their efforts of introducing functionality of these secure hardware to Java developers, we can better evaluate our solution based on the challenges and constraints they have been faced with.

7.1 Alternative to GP

There are many alternative TEEs available in current mobile devices but they are not available for ordinary developers including us. So we only address the TEE which is publicly accessable.

Trusty TEE & API On the Android platform, Google provides a bundle of software components called Trusty which is a TEE [15]. It allows Android processes to utilize the services provided by TAs running on top of the Trusty OS whose implementation is hidden from developers. The Trusty provides two sets of C APIs: one facilitates inter-communications between TAs and another one defines how user-space processes (CAs) can talk to the TAs in secure world. Similar to GP TEE APIs, they are also come with the form of C style with limited C++ support. The TAs are single-threaded and do not supply multithreading mechanisms.

JSR 177 JSR 177 [1] defines a collection of Java APIs to provide secure services to J2ME [24] enabled devices. J2ME is a micro version of Java platform, which targets for devices with limited computation and storage

capabilities. So JSR177 enables Java applications running on top of J2ME to integrate secure elements, such as secure execution and storage.

7.2 TPM from Java

Unlike the TEE which is more flexible, faster and widely deployed in the mobile devices, the Trusted Platform Module (TPM) is either a piece of hardware attached to the motherboard or a software implementation which have specific usages, such as providing secure storage, a true random number generator and cryptographic operations [26, 32]. It mainly targets the field of Personal Computers (PCs) (there is also an effort to adopt its usage to Android [25]). The interfaces to access its functionality are defined in the TCG Software Stack (TSS) specification [33] by the Trusted Computing Group(TCG). The TSS defines three vertical layers from top to bottom: Trusted Service Provider (TSP) layer, Trusted Core Service (TCS) layer and Trusted Device Driver Library (TDDL) layer. The functionality of TPM is exposed to applications via the TSP Interface (TSPI).

One project called jTSS Wrapper wraps the C TSPI of the TrouSerS [31] to Java interfaces [34]. TrouSerS is an open-source implementation of the TSS by IBM. Resemblances between their work and ours are that there are existing specifications which standardize the ways to use security hardware. Resemblances are, too, that these specifications are in the form of C programming language, which leads the efforts to map the C API to Java API thus relieving the burdens of handling native code for Java developers. However they used the SWIG [4] as the glue for C and Java while we directly deal with the Java Native Interface (JNI) which can allow java object manipulations from both directions. Later this project is included into another one called jTSS which is a full implementation of TSS layers purely in Java [5, 16]. Depending on how a GP-compliant TEE brings its functionality to normal developers in the future, it is unlikely that ordinary applications can directly use a kernel driver to communicate with the TEE just like TPM does. In addition, OT-J is mainly targeting the Android environment which does not allow direct access to the kernel drivers.

There are other projects which also try to make TPM available for Java developers while not conforming to the TSS specification, such as TPM/J [27].

TPM/J is intended to not comply with the TSS specification which are used for experiments and R&D purposes by developers and researchers where the TSS is not that important [27]. The data objects, such as commands and responses, transferred as byte streams via TPM driver are mapped to

corresponding Java objects in TMP/J as such easy the access to the field members of these data objects.

Based on previous efforts introduced above, the Java community proposed its own Java API which provides the identical functionality as the TSS specification. The API proposal is named as JSR321 [30]. Compared with the TSS, JSR321 also conform to the concepts introduced by the TCG. But it hides the underlying interactions with TPM driver and only expose the API with Java conventions to developers.

Chapter 8

Conclusion and Future Work

Normally the functionality of the TEE is hidden from ordinary developers. Thanks to the efforts of GlobalPlatform by publishing TEE related specifications, it makes a source-level compatibility between TEEs from different vendors. Similar to the usages of other secure hardware, the specifications define C API which can be problematic for Java developers. There are efforts trying to make the functionality of secure hardware available to Java developers as stated in chapter 7. Such efforts include providing a Java wrapper library on top of the C API or re-implementing the C API in Java. The approach we have taken is providing a Java wrapper library named as *OT-J* on top of the GP TEE Client API. To validate the functionality and usability of the OT-J, a prototype implementation has been done on top of Android architecture. We also evaluate the outcome of the OT-J to prove that our evaluation shows that OT-J satisfies the requirements initially defined in chapter 3.

In terms of future works, extra efforts should be put on integrating the features of OmniShare TA to OmniShare application.

Bibliography

[1] Ahmad, S. JSR 177 Security and Trust Services API for J2ME, 2004.

[2] ARM. ARM Security Technology - Building a Secure System using TrustZone Technology, April 2009. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[3] ARM, A. Security Technology Building a Secure System Using TrustZone Technology (white paper). *ARM Limited* (2009).

[4] Beazley, D. M., et al. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Tcl/Tk Workshop* (1996).

[5] Dietrich, K., Pirker, M., Vejda, T., Toegl, R., Winkler, T., and Lipp, P. A practical approach for establishing trust relationships between remote platforms using trusted computing. In *Trustworthy Global Computing*. Springer, 2007, pp. 156–168.

[6] Ekberg, J.-E., Kostiainen, K., and Asokan, N. The untapped potential of trusted execution environments on mobile devices. *IEEE Security & Privacy*, 4 (2014), 29–37.

[7] GlobalPlatform. TEE Client API Specification v1.0, July 2010.

[8] GlobalPlatform. TEE System Architecture v1.0, December 2011. http://www.globalplatform.org/specificationsdevice.asp.

[9] GlobalPlatform. TEE Internal Core API Specification v1.1, July 2014.

[10] GlobalPlatform Org. About GlobalPlatform. https://www.globalplatform.org/aboutus.asp. Accessed 20.05.2016.

[11] Google Inc. Android Keystore System. https://developer.android.com/training/articles/keystore.html.

[12] Google Inc. Android NDK. `https://developer.android.com/ndk/index.html`. Accessed 20.05.2016.

[13] Google Inc. Passing Objects over IPC. `https://developer.android.com/guide/components/aidl.html#PassingObjects`. Accessed 18.05.2016.

[14] Google Inc. Protocol buffers. `https://developers.google.com/protocol-buffers/`. Accessed 16.05.2016.

[15] Google Inc. Trusty TEE. `https://source.android.com/security/trusty/index.html`.

[16] Graz, T. IAIK. jTSS–Java TCG Software Stack, 2009.

[17] IETF. A Universally Unique IDentifier (UUID) URN Namespace. `https://www.ietf.org/rfc/rfc4122.txt`.

[18] Intel. Intel Software Guard Extensions. `https://software.intel.com/en-us/sgx`.

[19] Jens Wiklander. Generic TEE Subsystem, July 2015. `https://lwn.net/Articles/650487/`.

[20] Kostiainen, K., Ekberg, J.-E., Asokan, N., and Rantala, A. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (2009), ACM, pp. 104–115.

[21] McCarty, B. *Selinux: Nsa's open source security enhanced linux*. O'Reilly Media, Inc., 2004.

[22] McGillion, Brian and Dettenborn, Tanel and Nyman, Thomas and Asokan, N. Open-TEE An Open Virtual Trusted Execution Environment. In *Trustcom/BigDataSE/ISPA, 2015 IEEE* (2015), vol. 1, IEEE, pp. 400–407.

[23] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *HASP@ ISCA* (2013), p. 10.

[24] Muchow, J. W. *Core J2ME technology and MIDP*. Prentice Hall PTR, 2001.

[25] Othman, A. T., Khan, S., Nauman, M., and Musa, S. Towards a high-level trusted computing API for Android software stack. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication* (2013), ACM, p. 17.

[26] Perez, R., Sailer, R., van Doorn, L., et al. vTPM: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium* (2006), pp. 305–320.

[27] Sarmenta, L., Rhodes, J., and Müller, T. TPM/J Java-based API for the trusted platform module, 2007.

[28] Smalley, S., and Craig, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS* (2013), vol. 310, pp. 20–38.

[29] Tamrakar, Sandeep and Nguyen, Long and Pendyala, Praveen Kumar and Paverd, Andrew and Asokan, N and Sadeghi, Ahmad-Reza. OmniShare: Securely Accessing Encrypted Cloud Storage from Multiple Authorized Devices. *arXiv preprint arXiv:1511.02119* (2015).

[30] Toegl, R., et al. JSR 321: Trusted Computing API for Java. Java Community Process, 2008.

[31] Trousers, I. A TSS implementation for Linux, 2005.

[32] Trusted Computing Group. Trusted Platform Module (TPM) Summary. `http://www.trustedcomputinggroup.org/trusted-platform-module-tpm-summary/`. Accessed 23.05.2016.

[33] Trusted Computing Group. TCG Software Stack (TSS) Specification, Version 1.2, Errata A, March 2007. `http://www.trustedcomputinggroup.org/wp-content/uploads/TSS_1_2_Errata_A-final.pdf`.

[34] Winkler, T., et al. jTSS Wrapper for TSS 1.1 b stack. *The Institute for Applied Information Processing and Communications/Open Trusted Computing [online]*.

Appendix A

OT-J Documentation

Package
fi.aalto.ssg.opentee

Java API Version: V 1.0 beta

This is the main entrance of public APIs. In order to help explaining the APIs, there are several essential key words defined in the following:
1. CA: Client Application that the developer is creating;
2. TA: Trusted Application which is already deployed in TEE;
3. TEE: Trusted Execution Environment in target Android device within which TAs are running;
4. TEE (Proxy) Service Manager: Android service layer abstraction for TEE, which is responsible for handling incoming connections from CAs and for communicating with the TEE with the help of Native Libtee;
5. Native Libtee: A library which enables the communication between TEE and TEE Service Manager;
6. Underlying library: A library which resides in the CA and communicates with the remote TEE service on the behalf of CAs.

Introduction
This public API documentation defines the Java APIs corresponding to the C APIs defined in the GlobalPlatform Device Technology TEE Client API specification V 1.0. It describes how the CA should communicate with a remote TEE service manager.

Target audience
This document suits for software developers implementing:
1. Android version of CAs running within the rich operating environment, which needs to utilizing the functions of TAs;
2. TAs running inside the TEE which need to expose its internal functions to CAs.

Background information
1. what is TEE?
TEE stands for Trusted Execution Environment. There is another notation called Rich Execution Environment (REE). These two are often brought together to help explain both of them by comparisons. Before taking a look at TEE, it is better to start explaining from REE since it is more closer to our daily sense. REE represents the common operating system along with its hardware, such as devices running Windows, Mac OSX, Linux, Android or iOS. It abstracts the underlying hardware and provides resources for the applications to run with. As such, it has rich features for applications to utilize. However, the REE frequently suffer from different kinds of attacks, such as malware, worm, trojan and ransomware. In order to protect very sensitive and private information such as encryption private keys against these attacks, it is good to keep these private information safely in a separate container in case the REE is compromised. It is the similar notion as the safe deposit box. For instance, if bad guys broke into a home, it is still impossible for them to get all your money in the safe deposit box without the right password to open it. So, with such a thought, TEE showed up to meet such needs. Currently, the TEE shipped within devices is physically separated with REE by hardware boundaries. CAs run in the REE and TAs runs in the TEE. Compared with the rich features of REE, TEE mostly comes with very limited hardware capabilities.

2. GP Specification for TEE Client API Specification.
GP is short for GlobalPlatform. It is a non-profit organization that publishes specifications to promote security and interoperability of secure devices. One of the specifications it published, named "GlobalPlatform Device Technology TEE Client API Specification" (GP Client API), standardizes the ways how CAs communicate with TAs. GlobalPlatform also have other specifications for TEE but we only focus on this one specifically. The specification defines the C data types and functions for CAs to communicate with TAs.

3. Open-TEE
Open-TEE is an open virtual Trusted Execution Environment which conforms to the GP TEE Specifications. For devices which are not equipped with real hardware-based TEE, it can provide a virtual TEE for developers to debug and deploy TAs before shipping applications to a real TEE.

API Design
1. What are these Java APIs and what their relationships with GP TEE Client Specification?
In general, these APIs are the Java version of C APIs in GP Client API specification with a reformed design to fit Java development conventions, which mainly target on the Android devices. It can be used to develop Android CAs which want to utilize the functionality which TAs offer. It provides all the necessary functions for CAs to communicate with remote TAs just like the C APIs defined in GP Specification.

2. Why they are needed?

(continued on next page)

(continued from last page)

In GP TEE Client Specification, it only specify the C data types and APIs which limit or complicate the development of CAs which aim for Android devices. Since Java is the mainstream programming language to develop Android applications, for Android developers who wants to utilize the GP C API to enable the communications between CAs and TAs, it would be troublesome to deal with native code development, especially for those who are not familiar with it, which can result in more potential bugs and unexpected behaviours if not handled correctly. Under such circumstances, every developers have to re-write these codes with the similar functionality, which can be a waste of efforts and error-prone. To avoid such awkward situations, an open-sourced design, which can enable the CAs communicate with TAs while provide nice and clean public interfaces for Android developers, is urgent to conquer this issue. With such a thought, the coming public Java APIs are available for Android developers, which can release them from the burdens of dealing native development in Android. It might be not that efficient as directly dealing with C APIs but the performance should be in a tolerant level. In addition, all the implementations of public APIS are open-source for everyone. By taking feedback from developers, these shared codes can be more bug-free and efficient.

3. How to use it and what to expect from the APIs?
a. Prerequisites
- The TA is already deployed in Open-TEE.
- The Android application which provides a remote TEE Proxy service should be running.
b. Check the descriptions for each API.

Bug report to:
rui.yang at aalto.fi

Organization:
Security System Group, Aalto University School of Science, Espoo, Finland.

Appendix: Example code chapter
The following example codes demonstrate how to utilize the Java API to communicate with the TAs residing in the TEE.

Firstly, we assume that we get an `ITEEClient` interface by calling a factory method. The way to obtain an `ITEEClient` interface depends on real implementation. The following code is just an example.

```
ITEEClient client = FactoryMethodWrappers.newTEEClient();
```

Right now, we want to establish a connection to a remote TEE Proxy service so that we can interact with the TAs running inside of TEE. By using the `ITEEClient` interface we obtained from last step, we can establish a connection to a remote TEE by calling `initializeContext` method in `ITEEClient` interface. For the two input parameters, please refer to the API definition in `ITEEClient.IContext` interface. If no exception is caught, a valid `IContext` interface will be returned and program flow continues to next block of code. Otherwise, the returned `IContext` interface will be null and an exception will be threw. For different kinds of exceptions can be threw by this API, please also refer to this API definition in `ITEEClient.IContext` interface. In the handling exception code block, it is recommended to re-initializeContext again and the program flow should not continue until it get a valid `IContext` interface.

```
ITEEClient.IContext ctx = null;

final String param_TEE_NAME = null; // connect to the default TEE.
final android.context.Context param_app_context = getApplicationContext();

try {
    ctx = client.initializeContext(param_TEE_NAME,
                                   param_app_context);
} catch (TEEClientException e) {
    // handle TEEClientException here.
}
```

After we successfully connected to the remote TEE Proxy service, in order to interact with one TA, we must open a session to the TA by providing correct authentication data. To open a session, the function `openSession` within `IContext` interface must be called. For the input parameters for the API and possible exceptions threw by it, please refer to the API definition in `ITEEClient.IContext` interface. For the creation of param_operation parameter, please refer to the example code which creates

(continued on next page)

(continued from last page)

an `IOperation` interface using the factory method `newOperation` in the coming sections.

```
ITEEClient.ISession ses = null;

final UUID param_uuid = new UUID(0x1234567887654321L, 0x0102030405060708L);
final ConnectionMethod param_conn_method =
        ITEEClient.IContext.ConnectionMethod.LoginPublic;
final Integer param_conn_data = null;

try {
    ses = ctx.openSession(param_uuid,
                param_conn_method,
                param_conn_data,
                param_operation);
} catch (TEEClientException e) {
    // handle TEEClientException here.
}
```

After successfully opened a session to a specific TA, a valid `ITEEClient.ISession` interface will be returned. So we can interact with TA by using `invokeCommand` API within the `ITEEClient.ISession` interface. The creation of param_operation please also refer to the same example code which creates an `IOperation` interface.

```
final int param_comm_id = 0x12345678;

try{
    ses.invokeCommand(param_comm_id,
                      param_operation);
}catch (TEEClientException e) {
    // handle TEEClientException here.
}
```

In some cases, data is needed to be transferred between CAs and TAs. So the API provides two different kinds of data encapsulation mechanisms. After that, they can be encapsulated again within `ITEEClient.IOperation` which can be sent to TA during `openSession` or `invokeCommand` calls.

The first approach is to create an `ITEEClient.IValue` interface by calling `newValue` factory method in `ITEEClient`. So up to 2 integer values can be encapsulated. The two values are given when calling `newValue` function and further interactions with this pair of values are defined in the `ITEEClient.IValue` interface.

```
final ITEEClient.IValue.Flag param_flag = ITEEClient.IValue.Flag.TEEC_VALUE_INOUT;

int param_value_A = 66;
int param_value_B = 88;

ITEEClient.IValue val = client.newValue(param_flag,
                                        param_value_A,
                                        param_value_B);
```

Another approach to transfer the data is using shared memory. The notation shared memory in here works as follows. Firstly, CA create a byte array as the buffer for the shared memory. Then, the CA registers the byte array as a shared memory to the TA so that

(continued on next page)

(continued from last page)

TA can also operate on the buffer. To create a shared memory, the CA must call `registerSharedMemory` method in `ITEEClient.IContext`. An `ITEEClient.ISharedMemory` interface will be returned.

```
ITEEClient.ISharedMemory sm = null;

byte[] param_byte_array = new byte[256];

ITEEClient.ISharedMemory param_flags =
        ITEEClient.ISharedMemory.TEEC_MEM_INPUT |
        ITEEClient.ISharedMemory.TEEC_MEM_OUTPUT;

try{
    sm = ctx.registerSharedMemory(param_byte_array,
                                  param_flags);
} catch (TEEClientException e) {
    // handle TEEClientException here.
}
```

After encapsulating the data within `IValue` interface or `ISharedMemory` interface, in order to share the data with TA, we must encapsulate these interfaces again into an `ITEEClient.IOperation` interface which then can be passed to TA during `openSession` or `invokeCommand` calls. The `IValue` interface can be directly used. However, to use the shared memory, the `ISharedMemory` interface must be encapsulated again into an `ITEEClient.IRegisteredMemoryReference` interface. Then along with the `IValue` interface, it can be used to create an `ITEEClient.IOperation` interface. To create an `IRegisteredMemoryReference` interface, the factory method `newRegisteredMemoryReference` within `ITEEClient` must be called.

```
ITEEClient.IRegisteredMemoryReference.Flag param_flags =
        ITEEClient.IRegisteredMemoryReference.Flag.TEEC_MEMREF_INOUT;

final param_offset = 0;

ITEEClient.IRegisteredMemoryReference rmr =
        client.newRegisteredMemoryReference(sm,
                                            param_flags,
                                            param_offset);
```

To create an `IOperation` interface, the factory method `newOperation` within `ITEEClient` must be called. The input parameters can be up to 4 `IValue` or `IRegisteredMemoryReference` interfaces.

```
ITEEClient.IOperation op = client.newOperation(rmr, val);
```

Resource cleaning up

If shared memory is no longer needed, it must be released by calling `releaseSharedMemory` function within `IContext` interface.

(continued on next page)

(continued from last page)

```
try {
    ctx.releaseSharedMemory(sm);
} catch (TEEClientException e) {
    // handle TEEClientException here.
}
```

The session also must be closed if CA no longer wants to interact with the TA.

```
try {
    ses.closeSession();
} catch (TEEClientException e) {
    // handle TEEClientException here.
}
```

Once CA no longer need to communicate with TEE, the context must be finalized. Be aware to release all the resources, mainly shared memory, and close all sessions before finalizing context.

```
try {
    ctx.finalizeContext();
} catch (TEEClientException e) {
    // handle TEEClientException here.
}
```

fi.aalto.ssg.opentee
Interface ITEEClient

public interface **ITEEClient**
extends

Open-TEE Java API entry point. `ITEEClient` interface embraces all APIs and public interfaces. CA can use it to communicate with a remote TEE/TA. The way how an `ITEEClient` can be obtained is determined by real implementations. It is not specified in this Java API.

Nested Class Summary

class	`ITEEClient.IContext` ITEEClient.IContext
class	`ITEEClient.IOperation` ITEEClient.IOperation
class	`ITEEClient.IParameter` ITEEClient.IParameter
class	`ITEEClient.IRegisteredMemoryReference` ITEEClient.IRegisteredMemoryReference
class	`ITEEClient.ISession` ITEEClient.ISession
class	`ITEEClient.ISharedMemory` ITEEClient.ISharedMemory
class	`ITEEClient.IValue` ITEEClient.IValue
class	`ITEEClient.ReturnOriginCode` ITEEClient.ReturnOriginCode

Field Summary

public static final	`TEEC_SUCCESS` The return value for TEEC_SUCCESS. Value: **`0`**

Method Summary

abstract `ITEEClient.IContext`	`initializeContext(java.lang.String teeName, Context context)` A method which initializes a context to a TEE.
abstract `ITEEClient.IOperation`	`newOperation()` a method to create an operation without parameter.
abstract `ITEEClient.IOperation`	`newOperation(ITEEClient.IParameter firstParam)` a method to create an operation with one parameter.

abstract ITEEClient.IOperation	newOperation(ITEEClient.IParameter firstParam, ITEEClient.IParameter secondParam) a method to create an operation with two parameters.
abstract ITEEClient.IOperation	newOperation(ITEEClient.IParameter firstParam, ITEEClient.IParameter secondParam, ITEEClient.IParameter thirdParam) a method to create an operation with three parameters.
abstract ITEEClient.IOperation	newOperation(ITEEClient.IParameter firstParam, ITEEClient.IParameter secondParam, ITEEClient.IParameter thirdParam, ITEEClient.IParameter forthParam) a method to create an operation with four parameters.
abstract ITEEClient.IRegisteredMemoryReference	newRegisteredMemoryReference(ITEEClient.ISharedMemory sharedMemory, ITEEClient.IRegisteredMemoryReference.Flag flag, int offset) A method to create a IRegisteredMemoryReference interface with a valid ISharedMemory interface.
abstract ITEEClient.IValue	newValue(ITEEClient.IValue.Flag flag, int a, int b) A method to create an interface of a pair of two integer values.

Fields

TEEC_SUCCESS

```
public static final int TEEC_SUCCESS
```

The return value for TEEC_SUCCESS.
Constant value: **0**

Methods

newOperation

```
public abstract ITEEClient.IOperation newOperation()
```

a method to create an operation without parameter.

Returns:
an IOperation interface for created operation.

newOperation

```
public abstract ITEEClient.IOperation newOperation(ITEEClient.IParameter firstParam)
```

a method to create an operation with one parameter. It is possible to create multiple IOperation interfaces using the same IParameter. But it is not recommended especially when the I/O direction of IParameter is output for TA since it is possible that such an IParameter is in an inconsistent state. This rule also apply to other newOperation overloaded functions which take IParameter(s) as inputs.

Parameters:
firstParam - the first IParameter.

Returns:
an IOperation interface for created operation.

(continued on next page)

(continued from last page)

newOperation

```
public abstract ITEEClient.IOperation newOperation(ITEEClient.IParameter firstParam,
          ITEEClient.IParameter secondParam)
```

a method to create an operation with two parameters. The order of input parameters should be aligned with the order of required parameters in TA. This rule also apply to other overloaded `newOperation` functions which takes more than two parameters.

Parameters:

`firstParam` - the first `IParameter`.
`secondParam` - the second `IParameter`.

Returns:

an `IOperation` interface for created operation.

---

newOperation

```
public abstract ITEEClient.IOperation newOperation(ITEEClient.IParameter firstParam,
          ITEEClient.IParameter secondParam,
          ITEEClient.IParameter thirdParam)
```

a method to create an operation with three parameters.

Parameters:

`firstParam` - the first `IParameter`.
`secondParam` - the second `IParameter`.
`thirdParam` - the third `IParameter`.

Returns:

an `IOperation` interface for created operation.

---

newOperation

```
public abstract ITEEClient.IOperation newOperation(ITEEClient.IParameter firstParam,
          ITEEClient.IParameter secondParam,
          ITEEClient.IParameter thirdParam,
          ITEEClient.IParameter forthParam)
```

a method to create an operation with four parameters.

Parameters:

`firstParam` - the first `IParameter`.
`secondParam` - the second `IParameter`.
`thirdParam` - the third `IParameter`.
`forthParam` - the forth `IParameter`.

Returns:

an `IOperation` interface for created Operation.

---

newRegisteredMemoryReference

```
public abstract ITEEClient.IRegisteredMemoryReference
newRegisteredMemoryReference(ITEEClient.ISharedMemory sharedMemory,
          ITEEClient.IRegisteredMemoryReference.Flag flag,
          int offset)
  throws BadParametersException
```

A method to create a `IRegisteredMemoryReference` interface with a valid `ISharedMemory` interface. The flag parameter is only taken into considerations when the I/O direction(s) it implies are a subset of I/O directions of the referenced shared memory. It will not override the flags which the shared memory already have.

(continued on next page)

(continued from last page)

Parameters:

`sharedMemory` - the shared memory to refer.
`flag` - the flag for referenced shared memory.
`offset` - the offset from the beginning of the buffer of shared memory.

newValue

```
public abstract ITEEClient.IValue newValue(ITEEClient.IValue.Flag flag,
         int a,
         int b)
```

A method to create an interface of a pair of two integer values.

Parameters:

`flag` - The I/O directory of `IValue` for TAs.
`a` - The first integer value.
`b` - The second integer value.

Returns:

an `IValue` interface.

initializeContext

```
public abstract ITEEClient.IContext initializeContext(java.lang.String teeName,
         Context context)
  throws TEEClientException
```

A method which initializes a context to a TEE.

Parameters:

`teeName` - the name of remote TEE. If teeName is null, a context will be initialized within a default TEE.
`context` - Android application context.

Returns:

`IContext` interface.

Throws:

`exception.AccessDeniedException:` - Unable to initialize a context with the remote TEE due to insufficient privileges of the CA.
`exception.BadStateException:` - TEE is not ready to initialize a context for the CA.
`exception.BadParametersException:` - providing an invalid Android context.
`exception.BusyException:` - TEE is busy.
`exception.CommunicationErrorException:` - Communication with remote TEE service failed.
`exception.GenericErrorException:` - Non-specific cause exception.
`exception.TargetDeadException:` - TEE crashed.

fi.aalto.ssg.opentee
Interface ITEEClient.IOperation

public interface **ITEEClient.IOperation**
extends

This interface defines the way to interact with an `Operation` which is a wrapper class for 0 to 4 `IParameter`(s). It can be created only by calling the function `newOperation`. After a valid `IOperation` interface is returned, developers can refer to the corresponding `Operation` in either `openSession` or `invokeCommand` function calls. When dealing with multiple threads, one `IOperation` interface can be shared between different threads. So it is possible that multiple threads try to access the same `IOperation` interface at the same time. If one or more `IParameter` interfaces wrapped inside the `IOperation` is output for the TA, it is possible that the `IParameter`(s) might be in an inconsistent state which may result in an incorrect read of the corresponding wrapped resources within `IParameter`(s), such as `IValue` and `SharedMemory`. Furthermore, if one thread attempts to apply one `IOperation` interface in its `openSession` or `InvokeCommand` function call while this `IOperation` interface is being used by another thread, a `BusyException` will be thrown. In addition, if one `IOperation` interface is modified by another thread, it is the responsibilities of developers to be aware of the changes. In order to avoid misuse of the `IOperation` interface, developers should not access any wrapped resources in an `IOperation` interface which is in use. The state of the `IOperation` can be obtained by calling its `isStarted` function. So it is recommended that developers should check the state of the `IOperation` interface before accessing it.

Method Summary

`abstract boolean`	`isStarted()` If one `IOperation` interface is being used in an ongoing operation (either openSession or invokeCommand) in a separate thread, this function will return true.

Methods

isStarted

```
public abstract boolean isStarted()
```

If one `IOperation` interface is being used in an ongoing operation (either openSession or invokeCommand) in a separate thread, this function will return true. Developers can utilize this function to test the availability of the `IOperation` interface.

Returns:

true if `IOperation` is under usage. Otherwise false if not being used.

fi.aalto.ssg.opentee
Interface ITEEClient.IParameter

All Subinterfaces:
IValue, IRegisteredMemoryReference

public interface **ITEEClient.IParameter**
extends

`IParameter` interface is the super class of `IRegisteredMemoryReference` and `IValue` interfaces, It can passed into the `newOperation` to create an `IOperation` interface. It is possible to share the `IParameter` interface between different threads. Developers should be ware of the race condition when accessing the same `IParameter`. It is also their responsibilities to handle such a scenario.

Nested Class Summary

class	`ITEEClient.IParameter.Type` ITEEClient.IParameter.Type

Method Summary

abstract `ITEEClient.IParameter.Type`	`getType()` Get the type of the `IParameter` interface.

Methods

getType

`public abstract ITEEClient.IParameter.Type getType()`

Get the type of the `IParameter` interface.

Returns:
an enum value Type which can be either TEEC_PTYPE_VAL or TEEC_PTYPE_RMR.

fi.aalto.ssg.opentee
Class ITEEClient.IParameter.Type

```
java.lang.Object
   |
   +-java.lang.Enum
      |
      +-fi.aalto.ssg.opentee.ITEEClient.IParameter.Type
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Comparable

---

public static final class **ITEEClient.IParameter.Type**
extends java.lang.Enum

The enum to indicates the type of the parameter.

Field Summary

public static final	TEEC_PTYPE_RMR This Parameter is a RegisteredMemoryReference.
public static final	TEEC_PTYPE_VAL This Parameter is a Value.

Method Summary

static ITEEClient.IParameter.Type	valueOf(java.lang.String name)
static ITEEClient.IParameter.Type[]	values()

Methods inherited from class java.lang.Enum

clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface java.lang.Comparable

compareTo

Fields

(continued from last page)

TEEC_PTYPE_VAL

```
public static final fi.aalto.ssg.opentee.ITEEClient.IParameter.Type TEEC_PTYPE_VAL
```

This Parameter is a Value.

TEEC_PTYPE_RMR

```
public static final fi.aalto.ssg.opentee.ITEEClient.IParameter.Type TEEC_PTYPE_RMR
```

This Parameter is a RegisteredMemoryReference.

Methods

values

```
public static ITEEClient.IParameter.Type[] values()
```

valueOf

```
public static ITEEClient.IParameter.Type valueOf(java.lang.String name)
```

fi.aalto.ssg.opentee
Interface ITEEClient.IRegisteredMemoryReference

All Superinterfaces:
IParameter

public interface **ITEEClient.IRegisteredMemoryReference**
extends ITEEClient.IParameter

Interface for registered memory reference. When a shared memory needs to be passed to a remote TEE/TA, it must be wrapped within the `IRegisteredMemoryReference`. It can be only obtained by calling the `newRegisteredMemoryReference` function. It is possible that multiple `IRegisteredMemoryReference` interfaces are referencing the same `ISharedMemory` interface. So developers should be aware of such a situation

Nested Class Summary

class	`ITEEClient.IRegisteredMemoryReference.Flag` ITEEClient.IRegisteredMemoryReference.Flag

Method Summary

abstract int	`getOffset()` Get the offset set previously.
abstract int	`getReturnSize()` Get the size of returned buffer from TEE/TA.
abstract ITEEClient.ISharedMemory	`getSharedMemory()` Get the referenced registered shared memory.

Methods inherited from interface `fi.aalto.ssg.opentee.ITEEClient.IParameter`

`getType`

Methods

getSharedMemory

`public abstract ITEEClient.ISharedMemory getSharedMemory()`

Get the referenced registered shared memory.

Returns:
`ISharedMemory` interface for the referenced shared memory.

getOffset

`public abstract int getOffset()`

Get the offset set previously.

Returns:

(continued from last page)

an integer with a value ranging from 0 to the size of referenced shared memory.

getReturnSize

```
public abstract int getReturnSize()
```

Get the size of returned buffer from TEE/TA. This function will return a valid value (>= 0) only when the following two requirements are met at the same time:

- either TEEC_MEMREF_OUTPUT or TEEC_MEMREF_INOUT is marked as the flag of referenced shared memory;
- the referenced shared memory also can be used as output for TAs.

Otherwise, 0 will be returned. This function is normally called after the TA or TEE writes some data back to the referenced shared memory so that CA can know how big is the size of the returned data.

Returns:

an integer value as the returned size.

fi.aalto.ssg.opentee
Class ITEEClient.IRegisteredMemoryReference.Flag

```
java.lang.Object
   ┗-java.lang.Enum
      ┗-fi.aalto.ssg.opentee.ITEEClient.IRegisteredMemoryReference.Flag
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Comparable

---

public static final class **ITEEClient.IRegisteredMemoryReference.Flag**
extends java.lang.Enum

Flag enum indicates the I/O direction of the referenced registered shared memory for TAs.

Field Summary

public static final	TEEC_MEMREF_INOUT The I/O directions of the referenced registered shared memory are both input and output for TAs.
public static final	TEEC_MEMREF_INPUT The I/O direction of the referenced registered shared memory is input for TAs.
public static final	TEEC_MEMREF_OUTPUT The I/O direction of the referenced registered shared memory is output for TAs.

Method Summary

static ITEEClient.IRegisteredMemoryReference.Flag	valueOf(java.lang.String name)
static ITEEClient.IRegisteredMemoryReference.Flag[]	values()

Methods inherited from class java.lang.Enum

clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface java.lang.Comparable

compareTo

(continued from last page)

Fields

TEEC_MEMREF_INPUT

```
public static final fi.aalto.ssg.opentee.ITEEClient.IRegisteredMemoryReference.Flag
TEEC_MEMREF_INPUT
```

The I/O direction of the referenced registered shared memory is input for TAs.

TEEC_MEMREF_OUTPUT

```
public static final fi.aalto.ssg.opentee.ITEEClient.IRegisteredMemoryReference.Flag
TEEC_MEMREF_OUTPUT
```

The I/O direction of the referenced registered shared memory is output for TAs.

TEEC_MEMREF_INOUT

```
public static final fi.aalto.ssg.opentee.ITEEClient.IRegisteredMemoryReference.Flag
TEEC_MEMREF_INOUT
```

The I/O directions of the referenced registered shared memory are both input and output for TAs.

Methods

values

```
public static ITEEClient.IRegisteredMemoryReference.Flag[] values()
```

valueOf

```
public static ITEEClient.IRegisteredMemoryReference.Flag valueOf(java.lang.String
name)
```

fi.aalto.ssg.opentee
Interface ITEEClient.IValue

All Superinterfaces:
IParameter

public interface **ITEEClient.IValue**
extends ITEEClient.IParameter

Interface to access a pair of two integer values. It can be only obtained by calling the `newValue` method.

Nested Class Summary

class	`ITEEClient.IValue.Flag` ITEEClient.IValue.Flag

Method Summary

abstract int	`getA()` Get the first value.
abstract int	`getB()` Get the second value.

Methods inherited from interface `fi.aalto.ssg.opentee.ITEEClient.IParameter`

`getType`

Methods

getA

`public abstract int getA()`

Get the first value.

Returns:
an integer.

getB

`public abstract int getB()`

Get the second value.

Returns:
an integer.

fi.aalto.ssg.opentee
Class ITEEClient.IValue.Flag

```
java.lang.Object
   ┗-java.lang.Enum
      ┗-fi.aalto.ssg.opentee.ITEEClient.IValue.Flag
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Comparable

---

public static final class **ITEEClient.IValue.Flag**
extends java.lang.Enum

Flag enum indicates the I/O direction of Values for TAs.

Field Summary

public static final	TEEC_VALUE_INOUT The I/O directions for Value are both input and output for TAs.
public static final	TEEC_VALUE_INPUT The I/O direction for Value is input for TAs.
public static final	TEEC_VALUE_OUTPUT The I/O direction for Value is output for TAs.

Method Summary

static ITEEClient.IValue.Flag	valueOf(java.lang.String name)
static ITEEClient.IValue.Flag[]	values()

Methods inherited from class java.lang.Enum

clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface java.lang.Comparable

compareTo

Fields

(continued from last page)

TEEC_VALUE_INPUT

```
public static final fi.aalto.ssg.opentee.ITEEClient.IValue.Flag TEEC_VALUE_INPUT
```

The I/O direction for Value is input for TAs.

TEEC_VALUE_OUTPUT

```
public static final fi.aalto.ssg.opentee.ITEEClient.IValue.Flag TEEC_VALUE_OUTPUT
```

The I/O direction for Value is output for TAs.

TEEC_VALUE_INOUT

```
public static final fi.aalto.ssg.opentee.ITEEClient.IValue.Flag TEEC_VALUE_INOUT
```

The I/O directions for Value are both input and output for TAs.

Methods

values

```
public static ITEEClient.IValue.Flag[] values()
```

valueOf

```
public static ITEEClient.IValue.Flag valueOf(java.lang.String name)
```

fi.aalto.ssg.opentee
Interface ITEEClient.ISession

public interface **ITEEClient.ISession**
extends

For a CA to communicate with a TA within a TEE, a session must be opened between the CA and TA. To open a session, the CA must call `openSession` within a valid context. When a session is opened, an `ISession` interface will be returned. It contains all functions for the CA to communicate with the TA. Within this session, developers can call the `invokeCommand` function to invoke a function within the TA. When the session is no longer needed, the developers should close the session by calling `closeSession` function.

Method Summary

abstract void	`closeSession()` Close the connection to the remote TA.
abstract void	`invokeCommand(int commandId, ITEEClient.IOperation operation)` Sending a request to the connected TA with agreed commandId and parameters.

Methods

invokeCommand

```
public abstract void invokeCommand(int commandId,
          ITEEClient.IOperation operation)
  throws TEEClientException
```

Sending a request to the connected TA with agreed commandId and parameters. The parameters are encapsulated in the operation.

Parameters:

`commandId` - command identifier that is previously agreed with the TA. Based on the command id, CA can tell TA to perform a certain action. TA will know what to perform.
`operation` - a wrapper of parameters for the action to take.

Throws:

`exception.AccessConflictException:` - using shared resources which are occupied by another thread;
`exception.BadFormatException:` - providing incorrect format of parameters in operation;
`exception.BadParametersException:` - providing parameters with invalid content;
`exception.BusyException:` - 1. the TEE is busy working on something else and does not have the computation power to execute requested operation;
2. the referenced `IOperation` interface is being used by another thread.
`exception.CancelErrorException:` - the provided operation is invalid due to the cancellation from another thread;
`exception.CommunicationErrorException:` - 1. fatal communication error occurred in the remote TEE and TA side;
2. Communication with the TEE proxy service failed.
`exception.ExcessDataException:` - providing too much parameters in the operation.
`exception.ExternalCancelException:` - current operation cancelled by external signal in the CA, remote TEE or TA side.
`exception.GenericErrorException:` - non-specific error.
`exception.ItemNotFoundException:` - providing invalid reference to a registered shared memory.
`exception.NoDataException:` - required data are missing in the operation.
`exception.NotImplementedException:` - action mapped with this command id is not implemented in TA yet.
`exception.NotSupportedException:` - action mapped with this command id is not supported in TA.

(continued from last page)

`exception.OutOfMemoryException:` - the remote system runs out of memory.
`exception.OverflowException:` - an buffer overflow happened in the remote TEE or TA.
`exception.SecurityErrorException:` - incorrect usage of shared memory.
`exception.ShortBufferException:` - the provided output buffer is too short to hold the output.
`exception.TargetDeadException:` - the remote TEE or TA crashed.

closeSession

```
public abstract void closeSession()
  throws TEEClientException
```

Close the connection to the remote TA. When dealing with multi-threads, this function is recommended to be called with the same thread which opens this session.

Throws:

`exception.CommunicationErrorException:` - Communication with remote TEE service failed.
`exception.TargetDeadException:` - the remote TEE or TA crashed.

fi.aalto.ssg.opentee
Interface ITEEClient.ISharedMemory

public interface **ITEEClient.ISharedMemory**
extends

In order to enable data sharing between a CA and TEE/TA, the notation called shared memory has been introduced to avoid expensive memory copies. A shared memory is a block of memory resides in the CA and a TEE/TA can operate on it directly. But how effective the shared memory is depends on the real implementation on specific systems. To create a shared memory, the CA firstly allocate a buffer which can be used as a shared memory. Then, the CA calls the `registerSharedMemory` to register the buffer as a shared memory to the remote TEE so that the TA can also operate on it. When the CA tries to register a shared memory, the I/O direction of this shared memory must be provided along with the buffer of the shared memory. The I/O direction is a bit mask of TEEC_MEM_INPUT and TEEC_MEM_OUTPUT. Note that the I/O direction of this shared memory is for the remote TEE/TA. See the detailed explanation of these two flags in the field description. The size of the shared memory is the same as the buffer that it holds. When the CA successfully register this buffer as a shared memory with a flag of TEEC_MEM_INPUT, any modification on this buffer will be synced to the TEE/TA during each function call from the CA to the TEE. Similarly, if the shared memory is flagged with TEEC_MEM_OUTPUT, any modification of the shared memory from the TEE side will be synced back to the CA after each remote function call from the CA to the TEE.

`ISharedMemory` interface provides operations on the shared memory. It is only valid in a `IContext` interface. This interface can be only obtained by calling `registerSharedMemory` function. If the registered shared memory is not longer needed, developers should release it by calling `releaseSharedMemory` function. After the shared memory is released, the buffer it holds will not longer used as a shared memory. So, any modification on it will no longer be synced to the remote the TEE/TA.

Field Summary

`public static final`	`TEEC_MEM_INPUT` This value indicates the I/O direction of the shared memory is input for both TEE and TA. Value: **1**
`public static final`	`TEEC_MEM_OUTPUT` This value indicates the I/O direction of the shared memory is output for both TEE and TA. Value: **2**

Method Summary

`abstract byte[]`	`asByteArray()` Get the content of the shared memory.
`abstract int`	`getFlags()` Get the I/O direction of the shared memory.

Fields

TEEC_MEM_INPUT

`public static final int` **`TEEC_MEM_INPUT`**

This value indicates the I/O direction of the shared memory is input for both TEE and TA.
Constant value: **1**

(continued from last page)

TEEC_MEM_OUTPUT

```
public static final int TEEC_MEM_OUTPUT
```

This value indicates the I/O direction of the shared memory is output for both TEE and TA.
Constant value: **2**

Methods

getFlags

```
public abstract int getFlags()
```

Get the I/O direction of the shared memory.

Returns:
the flags of `ISharedMemory`.

---

asByteArray

```
public abstract byte[] asByteArray()
```

Get the content of the shared memory. This function returns a reference to the buffer that the shared memory holds.

Returns:
an byte array reference.

fi.aalto.ssg.opentee
Class ITEEClient.ReturnOriginCode

```
java.lang.Object
   ┗-java.lang.Enum
      ┗-fi.aalto.ssg.opentee.ITEEClient.ReturnOriginCode
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Comparable

---

public static final class **ITEEClient.ReturnOriginCode**
extends java.lang.Enum

A enum indicates the origin when an exception is threw. It can be obtained by calling `getReturnOrigin` of a caught exception. Developers can get a valid return origin only when the exceptions are threw by these two functions: `openSession` and `invokeCommand`. Otherwise, the return origin will be null.

Field Summary

public static final	TEEC_ORIGIN_API The exception is originated within the TEE Client API implementation.
public static final	TEEC_ORIGIN_COMMS The exception is originated within the underlying communications stack linking: 1.
public static final	TEEC_ORIGIN_TA The exception is originated within the TA.
public static final	TEEC_ORIGIN_TEE The exception is originated within the common TEE code.

Method Summary

static ITEEClient.ReturnOriginCode	valueOf(java.lang.String name)
static ITEEClient.ReturnOriginCode[]	values()

Methods inherited from class `java.lang.Enum`

`clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Methods inherited from interface `java.lang.Comparable`

`compareTo`

Fields

TEEC_ORIGIN_API

`public static final fi.aalto.ssg.opentee.ITEEClient.ReturnOriginCode` **`TEEC_ORIGIN_API`**

The exception is originated within the TEE Client API implementation.

TEEC_ORIGIN_COMMS

`public static final fi.aalto.ssg.opentee.ITEEClient.ReturnOriginCode` **`TEEC_ORIGIN_COMMS`**

The exception is originated within the underlying communications stack linking:
1. the CA with remote TEE Proxy service;
2. the TEE Proxy service with the TEE.

TEEC_ORIGIN_TEE

`public static final fi.aalto.ssg.opentee.ITEEClient.ReturnOriginCode` **`TEEC_ORIGIN_TEE`**

The exception is originated within the common TEE code.

TEEC_ORIGIN_TA

`public static final fi.aalto.ssg.opentee.ITEEClient.ReturnOriginCode` **`TEEC_ORIGIN_TA`**

The exception is originated within the TA.

Methods

values

`public static ITEEClient.ReturnOriginCode[]` **`values`**`()`

valueOf

`public static ITEEClient.ReturnOriginCode` **`valueOf`**`(java.lang.String name)`

fi.aalto.ssg.opentee
Interface ITEEClient.IContext

public interface **ITEEClient.IContext**
extends

`IContext` interface provides all the functions to interact with an initialized context in remote TEE. This interface is returned by the `initializeContext` function call. When a context is no longer needed, it should be closed by calling `finalizeContext`. When the `IContext` interface is passed into different threads, developers are responsible for providing thread-safe mechanism to avoid the conflict between different threads.

Nested Class Summary

class	`ITEEClient.IContext.ConnectionMethod` ITEEClient.IContext.ConnectionMethod

Method Summary

abstract void	`finalizeContext()` Finalizing the context and close the connection to the TEE after all sessions have been terminated and all shared memories have been released.
abstract `ITEEClient.ISession`	`openSession(java.util.UUID uuid, ITEEClient.IContext.ConnectionMethod connectionMethod, java.lang.Integer connectionData, ITEEClient.IOperation operation)` Open a session with a TA within the current context.
abstract `ITEEClient.ISharedMemory`	`registerSharedMemory(byte[] buffer, int flags)` Register a block of existing CA memory as a shared memory within.
abstract void	`releaseSharedMemory(ITEEClient.ISharedMemory sharedMemory)` Releases the Shared Memory which was previously obtained using `registerSharedMemory`.
abstract void	`requestCancellation(ITEEClient.IOperation operation)` Requests the cancellation of a pending open session or a command invocation operation.

Methods

finalizeContext

```
public abstract void finalizeContext()
  throws TEEClientException
```

Finalizing the context and close the connection to the TEE after all sessions have been terminated and all shared memories have been released. This function is recommended to be called at the end of the thread which initialized the context.

Throws:

`exception.CommunicationErrorException:` - Communication with remote TEE service failed.

(continued from last page)

registerSharedMemory

```
public abstract ITEEClient.ISharedMemory registerSharedMemory(byte[] buffer,
          int flags)
  throws TEEClientException
```

Register a block of existing CA memory as a shared memory within. When this function tries to register a buffer as a shared memory which is already used by another shared memory, this function will also return a valid `ISharedMemory` interface. The TEE will regard this buffer as two identical shared memory. This will cause problems such as an `MacInvalidException`. However, when a shared memory is released, the buffer it holds can be registered again as a new shared memory. For the CA, the new shared memory has the same buffer but it is identical for the TEE.

Parameters:

`buffer` - pre-allocated byte array which is to be shared.

`flags` - indicates I/O direction of this shared memory for TAs.

Throws:

`exception.BadParametersException:` - 1. try to register a null/empty buffer as a shared memory; 2. providing incorrect flag value.

`exception.BadStateException:` - TEE is not ready to register a shared memory.

`exception.BusyException:` - TEE is busy.

`exception.CommunicationErrorException:` - Communication with remote TEE service failed.

`exception.ExternalCancelException:` - Current operation is cancelled by external signal in TEE.

`exception.GenericErrorException:` - Non-specific causes error.

`exception.NoStorageSpaceException:` - Insufficient storage in TEE.

`exception.OutOfMemoryException:` - Insufficient memory in TEE.

`exception.OverflowException:` - Buffer overflow in TEE.

`exception.TargetDeadException:` - TEE/TA crashed.

releaseSharedMemory

```
public abstract void releaseSharedMemory(ITEEClient.ISharedMemory sharedMemory)
  throws TEEClientException
```

Releases the Shared Memory which was previously obtained using `registerSharedMemory`. As stated in the description of the `ISharedMemory` interface, when the shared memory is released, the TEE/TA will no longer be able to read or write data to the shared memory. However, the buffer that this shared memory holds will still remain valid. When using the same shared memory within multi-threads, it is recommended to release the shared memory in the same thread which registered it.

Parameters:

`sharedMemory` - the reference to an `ISharedMemory` instance.

Throws:

`exception.CommunicationErrorException:` - Communication with the remote TEE service failed.

`exception.BadParametersException:` - Incorrect `ISharedMemory` instance such as passing a null object.

openSession

```
public abstract ITEEClient.ISession openSession(java.util.UUID uuid,
          ITEEClient.IContext.ConnectionMethod connectionMethod,
          java.lang.Integer connectionData,
          ITEEClient.IOperation operation)
  throws TEEClientException
```

Open a session with a TA within the current context. A session is a channel through which a CA can communicate with a specific TA (specified by the uuid). In order to open such a channel successfully, the CA must provide precise and correct data to authenticate itself to the TA.

Parameters:

`uuid` - UUID of the TA.

`connectionMethod` - the method of connection to use.

(continued from last page)

`connectionData` - any necessary data for connectionMethod.
`operation` - operation to perform.

Returns:

an `ISession` interface.

Throws:

`exception.AccessDeniedException:` - Insufficient privilege.
`exception.BadFormatException:` - Using incorrect format of parameter(s).
`exception.BadParametersException:` - Unexpected value(s) for parameter(s).
`exception.BadStateException:` - TEE is not ready to open a session or the referenced IOperation interface is occupied by another thread.
`exception.BusyException:` - TEE is busy.
`exception.CancelErrorException:` - Current operation is cancelled by another thread.
`exception.CommunicationErrorException:` - Communication with remote TEE service failed.
`exception.ExternalCancelException:` - Cancelled by external interrupt.
`exception.GenericErrorException:` - Non-specific cause.
`exception.ItemNotFoundException:` - Referred shared memory not found.
`exception.NoDataException:` - Extra data expected.
`exception.NoStorageSpaceException:` - Insufficient data storage in TEE.
`exception.OutOfMemoryException:` - TEE runs out of memory.
`exception.OverflowException:` - Buffer overflow in TEE.
`exception.SecurityErrorException:` - Incorrect usage of shared memory.
`exception.ShortBufferException:` - the provided output buffer is too short to hold the output.
`exception.TargetDeadException:` - TEE/TA crashed.

requestCancellation

```
public abstract void requestCancellation(ITEEClient.IOperation operation)
  throws TEEClientException
```

Requests the cancellation of a pending open session or a command invocation operation. This can be called from a different thread from that which is waiting for the `IOperation` interface. It is not guaranteed that the operation can be cancelled.

Parameters:

`operation` - the started or pending operation instance.

Throws:

`exception.CommunicationErrorException:` - Communication with remote TEE service failed.

fi.aalto.ssg.opentee
Class ITEEClient.IContext.ConnectionMethod

```
java.lang.Object
   ┗-java.lang.Enum
      ┗-fi.aalto.ssg.opentee.ITEEClient.IContext.ConnectionMethod
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Comparable

---

public static final class **ITEEClient.IContext.ConnectionMethod**
extends java.lang.Enum

Connection Method enum with fixed value corresponding to GP specification when calling `openSession`.

Field Summary

public static final	`LoginApplication` Login data about the running CA process itself is provided.
public static final	`LoginGroup` Login data about the group running the CA process is provided.
public static final	`LoginGroupApplication` Login data about the group running the CA and about the Client Application and the about the CA itself is provided.
public static final	`LoginPublic` No login data is provided.
public static final	`LoginUser` Login data about the user running the CA process is provided.
public static final	`LoginUserApplication` Login data about the user running the CA and about the Client Application itself is provided.

Method Summary

static ITEEClient.IContext.ConnectionMethod	`valueOf(java.lang.String name)`
static ITEEClient.IContext.ConnectionMethod[]	`values()`

Methods inherited from class `java.lang.Enum`

`clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf`

Methods inherited from class `java.lang.Object`

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface `java.lang.Comparable`
compareTo

Fields

LoginPublic

```
public static final fi.aalto.ssg.opentee.ITEEClient.IContext.ConnectionMethod
LoginPublic
```

No login data is provided.

LoginUser

```
public static final fi.aalto.ssg.opentee.ITEEClient.IContext.ConnectionMethod
LoginUser
```

Login data about the user running the CA process is provided.

LoginGroup

```
public static final fi.aalto.ssg.opentee.ITEEClient.IContext.ConnectionMethod
LoginGroup
```

Login data about the group running the CA process is provided.

LoginApplication

```
public static final fi.aalto.ssg.opentee.ITEEClient.IContext.ConnectionMethod
LoginApplication
```

Login data about the running CA process itself is provided.

LoginUserApplication

```
public static final fi.aalto.ssg.opentee.ITEEClient.IContext.ConnectionMethod
LoginUserApplication
```

Login data about the user running the CA and about the Client Application itself is provided.

LoginGroupApplication

```
public static final fi.aalto.ssg.opentee.ITEEClient.IContext.ConnectionMethod
LoginGroupApplication
```

Login data about the group running the CA and about the Client Application and the about the CA itself is provided.

Methods

(continued from last page)

values

public static ITEEClient.IContext.ConnectionMethod[] **values**()

---

valueOf

public static ITEEClient.IContext.ConnectionMethod **valueOf**(java.lang.String name)

fi.aalto.ssg.opentee
Class OpenTEE

```
java.lang.Object
   |
   +-fi.aalto.ssg.opentee.OpenTEE
```

public class **OpenTEE**
extends java.lang.Object

Factory method wrapper for OpenTEE Android.

Constructor Summary

public	OpenTEE()

Method Summary

static ITEEClient	newTEEClient() Factory method to create a new TEEClient interface.

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

OpenTEE

`public OpenTEE()`

Methods

newTEEClient

`public static ITEEClient newTEEClient()`

Factory method to create a new TEEClient interface.

Returns:
an ITEEClient interface.

(a) In English

Figure A.1: Aalto logo variants