GLOBALPLATFORM®

THE STANDARD FOR MANAGING APPLICATIONS ON SECURE CHIP TECHNOLOGY

GlobalPlatform Device Technology

TEE Internal API Specification

Version 1.0

Public Release

December 2011

Document Reference: GPD_SPE_010

This page intentionally left blank.

Contents

Figures

Tables

1 Introduction

This specification defines a set of C APIs for the development of ***Trusted Applications*** *(TAs)* running inside a ***Trusted Execution Environment*** *(TEE)*. For the purposes of this document a TEE is expected to meet the requirements defined in the GlobalPlatform TEE System Architecture [2] specification, i.e., it is accessible from a ***Rich Execution Environment*** *(REE)* through the GlobalPlatform TEE Client API (described in GlobalPlatform TEE Client API Specification [1]) but is specifically protected against malicious attacks and only runs code trusted in integrity and authenticity.

The APIs defined in this document target the C language and provide the following set of functionalities to TA developers:

- Basic OS-like functionalities, such as memory management, timer, and access to configuration properties
- Communication means with ***Client Applications*** *(CAs)* running in the Rich Execution Environment
- Trusted Storage facilities
- Cryptographic facilities
- Time management facilities

The scope of this document is the development of Trusted Applications in the C language and their interactions with the TEE Client API [1]. It does not cover other possible language bindings or the run-time installation and management of Trusted Applications.

1.1 Audience

This document is suitable for software developers implementing Trusted Applications running inside the TEE which need to expose an externally visible interface to Client Applications and to use resources made available through the TEE Internal API, such as cryptographic capabilities and Trusted Storage.

This document is also intended for implementers of the TEE itself, its ***Trusted OS***, ***Trusted Core Framework***, the TEE APIs, and the communications infrastructure required to access Trusted Applications.

1.2 IPR Disclaimer

GlobalPlatform draws attention to the fact that claims that compliance with this specification may involve the use of a patent or other intellectual property right (collectively, “IPR”) concerning this specification may be published at https://www.globalplatform.org/specificationsipdisclaimers.asp. GlobalPlatform takes no position concerning the evidence, validity, and scope of these IPR claims.

1.3 Normative References

Table 1-1: Normative References

Standard / Specification	Description	Ref
GPD_SPE_007	GlobalPlatform Device Technology TEE Client API Specification	[1]
GPD_SPE_009	GlobalPlatform Device Technology TEE System Architecture	[2]
ISO/IEC 9899:1999	Programming languages – C	[3]
RFC 4122	A Universally Unique IDentifier (UUID) URN Namespace	[4]
RFC 2119	Key words for use in RFCs to Indicate Requirement Levels	[5]
RFC 2045	Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies	[6]

1.4 Terminology and Definitions

Table 1-2: Terminology and Definitions

Term	Definition
Cancellation Flag	An indicator that a Client has requested cancellation of an operation.
Client	Either of the following: • a Client Application using the TEE Client API • a Trusted Application acting as a client of another Trusted Application, using the Internal Client API
Client Application (CA)	An application running outside of the Trusted Execution Environment making use of the TEE Client API to access facilities provided by Trusted Applications inside the Trusted Execution Environment. Contrast *Trusted Application (TA)*.
Client Properties	A set of properties associated with the Client of a Trusted Application.
Command	A message (including a Command Identifier and four Operation Parameters) send by a Client to a Trusted Application to initiate an operation.
Command Identifier	A 32-bit integer identifying a Command.
Cryptographic Key Object	An object containing key material.
Cryptographic Key-Pair Object	An object containing material associated with both keys of a key-pair.
Cryptographic Operation Handle	An opaque reference that identifies a particular cryptographic operation.
Cryptographic Operation Key	The key to be used for a particular operation.
Data Object	An object containing a data stream but no key material.
Data Stream	Data associated with a persistent object (excluding Object Attributes and metadata).

Term	Definition
Implementation	A particular implementation of the Trusted OS.
Initialized Object	A transient object whose attributes have been populated.
Instance	A particular execution of a Trusted Application, having physical memory space that is separated from the physical memory space of all other TA instances.
Key Size	The key size associated with a Cryptographic Object; values are limited by the key algorithm used.
Key Usage Flags	Indicators of the operations permitted with a Cryptographic Object.
Metadata	Additional data associated with a Cryptographic Object: Key Size and Key Usage Flags.
Multi Instance Trusted Application	Denotes a Trusted Application for which each session opened by a client is directed to a separate TA instance.
Object Attribute	Small amounts of data used to store key material in a structured way.
Object Handle	An opaque reference that identifies a particular object.
Object Identifier	A variable-length binary buffer identifying a persistent object.
Operation Parameter	One of four data items passed in a Command, which can contain integer values or references to client-owned shared memory blocks.
Panic	An exception that kills a whole TA instance as a result of calling one of the API functions.
Parameter Annotation	Denotes the pattern of usage of a function parameter or pair of function parameters.
Persistent Object	An object identified by an Object Identifier and including a Data Stream. Contrast *Transient Object*.
Property	An immutable value identified by a name.
Property Set	Any of the following: • The configuration properties of a Trusted Application • Properties associated with a Client Application by the Rich Execution Environment • Properties describing characteristics of a TEE Implementation.
REE Time	A time value that is as trusted as the REE.
Rich Execution Environment (REE)	An environment that is provided and governed by a Rich OS, potentially in conjunction with other supporting operating systems and hypervisors; it is outside of the TEE. This environment and applications running on it are considered un-trusted. Contrast *Trusted Execution Environment (TEE)*.

Term	Definition
Rich OS	Typically an OS providing a much wider variety of features than that of the OS running inside the TEE. It is very open in its ability to accept applications. It will have been developed with functionality and performance as key goals, rather than security. Due to the size and needs of the Rich OS it will run in an execution environment outside of the TEE hardware (often called an REE – Rich Execution Environment) with much lower physical security boundaries. From the TEE viewpoint, everything in the REE has to be considered un-trusted, though from the Rich OS point of view there may be internal trust structures. Contrast *Trusted OS*.
Session	Logically connects multiple commands invoked on a Trusted Application.
Single Instance Trusted Application	Denotes a Trusted Application for which all sessions opened by clients are directed to a single TA instance.
Storage Identifier	A 32-bit identifier for a Trusted Storage Space that can be accessed by a Trusted Application.
System Time	A time value that can be used to compute time differences and operation deadlines.
TA Persistent Time	A time value set by the Trusted Application that persists across platform reboots and whose level of trust can be queried.
Task	The entity that executes any code executed in a Trusted Application.
Transient Object	An object containing attributes but no data stream, which is reclaimed when closed or when the TA instance is destroyed. Contrast *Persistent Object*.
Trusted Application (TA)	An application running inside the Trusted Execution Environment that provides security related functionality to Client Applications outside of the TEE or to other Trusted Applications inside the Trusted Execution Environment. Contrast *Client Application (CA)*.
Trusted Application Configuration Properties	A set of properties associated with the installation of a Trusted Application.
Trusted Core Framework or "Framework"	The part of the Trusted OS responsible for implementing the Trusted Core Framework API[1] that provides OS-like facilities to Trusted Applications and a way for the Trusted OS to interact with the Trusted Applications.

[1] The Trusted Core Framework API is described in Chapter 4.

Term	Definition
Trusted Execution Environment (TEE)	An execution environment that runs alongside but isolated from an REE. A TEE has security capabilities and meets certain security-related requirements: It protects TEE assets from general software attacks, defines rigid safeguards as to data and functions that a program can access, and resists a set of defined threats. There are multiple technologies that can be used to implement a TEE, and the level of security achieved varies accordingly. Contrast *Rich Execution Environment (REE).*
Trusted OS	An operating system running in the TEE providing the TEE Internal API to Trusted Applications.
Trusted Storage Spaces	Storage spaces accessible only to Trusted Applications.
Uninitialized Object	A transient object allocated with a certain object type and maximum size but with no attributes.
Universally Unique Identifier (UUID)	An identifier as specified in RFC 4122 [4].

1.5 Abbreviations and Notations

Table 1-3: Abbreviations

Term	Definition
AAD	Additional Authenticated Data
AE	Authenticated Encryption
AES	Advanced Encryption Standard
API	Application Programming Interface
CA	Client Application
CMAC	Cipher-based MAC
CRT	Chinese Remainder Theorem
CTS	CipherText Stealing
DES	Data Encryption Standard
DH	Diffie-Hellman
DSA	Digital Signature Algorithm
ETSI	European Telecommunications Standards Institute
HMAC	Hash-based Message Authentication Code
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IPR	Intellectual Property Rights
ISO	International Organization for Standardization

Term	Definition
IV	Initialization Vector
MAC	Message Authentication Code
MD5	Message Digest 5
MGF	Mask Generating Function
NIST	National Institute of Standards and Technology
OAEP	Optimal Asymmetric Encryption Padding
OS	Operating System
PKCS	Public Key Cryptography Standards
PSS	Probabilistic Signature Scheme
REE	Rich Execution Environment
RFC	Request For Comments; may denote a memorandum published by the IETF
RSA	Rivest, Shamir, Adleman
SHA	Secure Hash Algorithm
TA	Trusted Application
TEE	Trusted Execution Environment
UTC	Coordinated Universal Time
UTF	Unicode Transformation Format
UUID	Universally Unique Identifier
XTS	XEX-based Tweaked Codebook mode with ciphertext stealing (CTS)

1.6 Revision History

Table 1-4: Revision History

Date	Version	Description
December 2011	1.0	Initial Release.

2 Overview of the TEE Internal API

This specification defines a set of C APIs for the development of ***Trusted Applications*** *(TAs)* running inside a ***Trusted Execution Environment*** *(TEE)*. For the purposes of this document a TEE is expected to meet the requirements defined in the GlobalPlatform TEE System Architecture [2] specification, i.e., it is accessible from a ***Rich Execution Environment*** *(REE)* through the GlobalPlatform TEE Client API [1] but is specifically protected against malicious attacks and runs only code trusted in integrity and authenticity.

A TEE provides the Trusted Applications an execution environment with defined security boundaries, a set of security enabling capabilities, and means to communicate with ***Client Applications*** running in the Rich Execution Environment. This document specifies how to use these capabilities and communication means for Trusted Applications developed using the C programming language. It does not cover how Trusted Applications are installed or managed and does not cover other language bindings.

2.1 Trusted Applications

A Trusted Application (TA) is a program that runs in a Trusted Execution Environment (TEE) and exposes security services to its Clients.

A Trusted Application is command-oriented. Clients access a Trusted Application by opening a session with the Trusted Application and invoking commands within the session. When a Trusted Application receives a command, it parses the messages associated with the command, performs any required processing, and then sends a response back to the client.

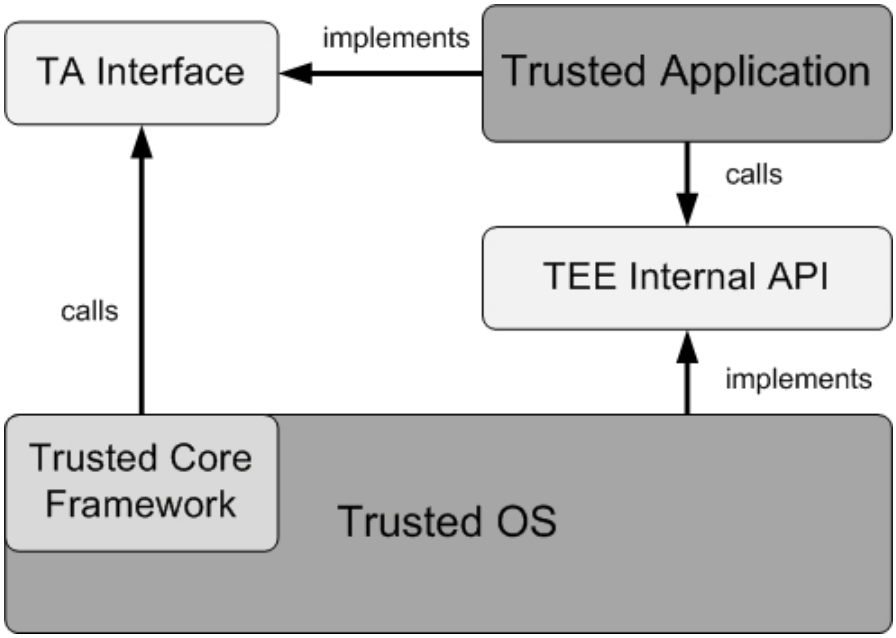
A Client typically runs in the Rich Execution Environment and communicates with a Trusted Application using the TEE Client API [1]. It is then called a "***Client Application***". It is also possible for a Trusted Application to act as a client of another Trusted Application, using the Internal Client API (see section 4.9). The term "***Client***" covers both cases.

2.1.1 TA Interface

Each Trusted Application exposes an interface (the TA interface) composed of a set of entry point functions that the Trusted Core Framework implementation calls to inform the TA about life-cycle changes and to relay communication between Clients and the TA. Once the Trusted Core Framework has called one of the TA entry points, the TA can make use of the TEE Internal API to access the facilities of the Trusted OS, as illustrated in Figure 2-1. For more information on the TA interface, see section 4.3.

Each Trusted Application is identified by a ***Universally Unique Identifier*** (UUID) as specified in RFC 4122 [4]. Each Trusted Application also comes with a set of Trusted Application Configuration Properties. These properties are used to configure the Trusted OS facilities exposed to the Trusted Application. Properties can also be used by the Trusted Application itself as a means of configuration.

Figure 2-1: Trusted Application Interactions with the Trusted OS



2.1.2 Instances, Sessions, Tasks, and Commands

When a Client creates a session with a Trusted Application, it connects to an ***Instance*** of that Trusted Application. A Trusted Application instance has physical memory space which is separated from the physical memory space of all other Trusted Application instances. The Trusted Application instance memory space holds the Trusted Application instance heap and writable global and static data.

All code executed in a Trusted Application is said to be executed by ***Tasks***. A Task keeps a record of its execution history (typically realized with a stack) and current execution state. This record is collectively called a Task context. A Task MUST be created each time the Trusted OS calls an entry point of the Trusted Application. Once the entry point has returned, an Implementation may recycle a Task to call another entry point but this MUST appear like a completely new Task was created to call the new entry point.

A ***Session*** is used to logically connect multiple commands invoked in a Trusted Application. Each session has its own state, which typically contains the session context and the context(s) of the Task(s) executing the session.

A ***Command*** is issued within the context of a session and contains a ***Command Identifier***, which is a 32-bit integer, and four ***Operation Parameters***, which can contain integer values or references to client-owned shared memory blocks.

It is up to the Trusted Application to define the combinations of commands and their parameters that are valid to execute. This is outside the scope of this specification.

2.1.3 Sequential Execution of Entry Points

All entry point calls within a given Trusted Application instance are called in sequence, i.e., no more than one entry point is executed at any point in time. The Trusted Core Framework implementation MUST guarantee that a commenced entry point call is completed before any new entry point call is allowed to begin execution.

If there is more than one entry point call to complete at any point in time, all but one call MUST be queued by the Framework. The order in which the Framework queues and picks enqueued calls for execution is implementation-defined.

It is not possible to execute multiple concurrent commands within a session. The TEE guarantees that a pending command has completed before a new command is executed.

Since all entry points of a given Trusted Application instance are called in sequence, there is no need to use any dedicated synchronization mechanisms to maintain consistency of any Trusted Application instance memory. The sequential execution of entry points inherently guarantees this consistency.

2.1.4 Cancellations

Clients can request the cancellation of open-session and invoke-command operations at any time.

If an operation is requested to be cancelled and has not reached the Trusted Application yet but has been queued, then the operation is simply retired from the queue.

If the operation has already been transmitted to the Trusted Application, then the task running the operation is put in the cancelled state. This has an effect on a few “cancellable” functions, such as `TEE_Wait`, but this effect may also be masked by the Trusted Application if it does not want to be affected by client cancellations. See section 4.10 for more details on how a Trusted Application can handle cancellation requests and mask their effect.

2.1.5 Unexpected Client Termination

When the client of a Trusted Application dies or exits abruptly and when it can be properly detected, then this must appear to the Trusted Application as if the client requests cancellation of all pending operations and gracefully closes all its client sessions. It must be indistinguishable from a clean session closing.

More precisely, the REE SHOULD detect when a Client Application dies or exits. When this happens, the REE MUST initiate a termination process that MUST result in the following sequence of events for all Trusted Application instances that are serving a session with the terminating client:

- If an operation is pending in the closing session, it must appear as if the client had requested its cancellation.
- When no operation remains pending in the session, the session must be closed.

If a TA client is a TA itself, this sequence of events MUST happen when the client TA panics.[2]

2.1.6 Instance Types

At least two Trusted Application instance types MUST be supported: Multi Instance and Single Instance. Whether a Trusted Application is Multi Instance or Single Instance is part of its configuration properties and MUST be enforced by the Trusted OS. See section 4.5 for more information on configuration properties.

- For a ***Multi Instance Trusted Application***, each session opened by a client is directed to a separate Trusted Application instance, created on demand when the session is opened and destroyed when the session closes. By definition, every instance of such a Trusted Application accepts and handles one and only one session at a given time.
- For a ***Single Instance Trusted Application***, all sessions opened by the clients are directed to a single Trusted Application instance. From the Trusted Application point of view, all sessions share the same Trusted Application instance memory space, which means for example that memory dynamically allocated for one session is accessible in all other sessions. It is also configurable whether a Single Instance Trusted Application accepts multiple concurrent sessions or not.

2.1.7 Configuration, Development, and Management

Trusted Applications as discussed in this document are developed using the C language. The way Trusted Applications are compiled and linked is implementation-dependent.

The way Trusted Applications are configured and installed in a TEE is also implementation-dependent. The scope of this specification does not include configuration, installation, de-installation, signing, verification, or any other life-cycle or deployment aspects.

[2] Panics are discussed in section 2.2.3.

2.2 Error Handling

2.2.1 Normal Errors

The TEE Internal API functions usually return an error code of type `TEE_Result` to indicate errors to the caller. This is used to denote "normal" run-time errors that the TA code is expected to catch and handle, such as out-of-memory conditions or short buffers.

2.2.2 Programmer Errors

There are a number of conditions in this specification that can only occur as a result of Programmer Error, i.e., they are triggered by incorrect use of the API by a Trusted Application, such as wrong parameters, wrong state, invalid pointers, etc., rather than by run-time errors such as out-of-memory conditions.

Some Programmer Errors are explicitly tagged as "Panic Reasons" and MUST be reliably detected by an ***Implementation***. These errors make it impossible to produce the result of the function and require that the API panic the calling TA instance, which kills the instance. If such a Panic Reason occurs, it MUST NOT go undetected and, e.g., produce incorrect results or corrupt TA data.

However, it is accepted that some Programmer Errors cannot be realistically detected at all times and that precise behavior cannot be specified without putting too much of a burden on the implementation. In case of such a Programmer Error, an Implementation is therefore not required to gracefully handle the error or even to behave consistently, but the Implementation SHOULD still make a best effort to detect the error and panic the calling TA. In any case, a Trusted Application MUST NOT be able to use a Programmer Error on purpose to circumvent the security boundaries enforced by an Implementation.

In general, incorrect handles—i.e., handles not returned by the API, already closed, with the wrong owner, type, or state—are definite Panic Reasons while incorrect pointers are imprecise Programmer Errors.

2.2.3 Panics

A ***Panic*** is an instance-wide uncatchable exception that kills a whole TA instance as a result of calling one of the API functions. It happens when the Implementation can detect a Programmer Error and also when the Trusted Application itself requests to panic by calling the function `TEE_Panic`.

When a Panic occurs, the Trusted Core Framework kills the panicking TA instance and does the following:

- It discards all client entry point calls queued on the TA instance and closes all sessions opened by Clients.
- It closes all resources that the TA instance opened, including all handles and all memory, and destroys the instance. Note that multiple instances can reference a common resource, for example an object. If an instance sharing a resource is destroyed, the Framework does not destroy the shared resource immediately, but will wait until no other instances reference the resource before reclaiming it.

After a Panic, no TA function of the instance is ever called again, not even `TA_DestroyEntryPoint`.

From the client's point of view, when a Trusted Application panics, the client commands must return the error `TEE_ERROR_TARGET_DEAD` with the origin `TEE_ORIGIN_TEE` until the session is closed. (For details about return origins, see the function `TEE_InvokeTACommand` in section 4.9.3 or the function `TEEC_InvokeCommand` in the TEE Client API Specification [1], §4.5.9.)

When a Panic occurs, an Implementation in a non-production environment, such as in a development or pre-production state, is encouraged to issue precise diagnostic information to help the developer understand the Programmer Error. Diagnostic information SHOULD NOT be exposed outside of a secure development environment.

2.3 Opaque Handles

This specification makes use of handles that opaquely refer to objects created by the API Implementation for a particular TA instance. A handle is only valid in the context of the TA instance that creates it and MUST always be associated with a type.

The special value `TEE_HANDLE_NULL`, which MUST always be `0`, is used to denote the absence of a handle. It is typically used when an error occurs or sometimes to trigger a special behavior in some function. For example, the function `TEE_SetOperationKey` clears the operation key if passed `TEE_HANDLE_NULL`. In general, the "close"-like functions do nothing if they are passed the NULL handle.

Other than the particular case of `TEE_HANDLE_NULL`, this specification does not define any constraint on the actual value of a handle.

Passing an invalid handle, i.e., a handle not returned by the API, already closed, or of the wrong type, is always a Programmer Error, except sometimes for the specific value `TEE_HANDLE_NULL`. When a handle is dereferenced by the API, the Implementation must always check its validity and panic the TA instance if it is not valid.

This specification defines a C type for each high-level type of handle. The following types are defined:

Table 2-1: Handle Types

Handle Type	Handle Purpose
`TEE_TASessionHandle`	Handle on sessions opened by a TA on another TA
`TEE_PropSetHandle`	Handle on a property set or a property enumerator
`TEE_ObjectHandle`	Handle on a cryptographic object
`TEE_ObjectEnumHandle`	Handle on a persistent object enumerator
`TEE_OperationHandle`	Handle on a cryptographic operation

These C types are defined as pointers on undefined structures. For example, `TEE_TASessionHandle` is defined as `struct __TEE_TASessionHandle*`. This is just a means to leverage the C language type-system to help separate different handle types. It does not mean that an Implementation has to define the structure, and handles do not need to represent addresses.

2.4 Properties

This specification makes use of ***Properties*** to represent configuration parameters, permissions, or implementation characteristics.

A property is an immutable value identified by a name, which is a Unicode string. The property value can be retrieved in a variety of formats: Unicode string, binary block, 32-bit integer, Boolean, and Identity.

Property names and values are intended to be rather small with a few hundreds of characters at most, although the specification defines no limit on the size of names or values.

In this specification, Unicode strings are always encoded in zero-terminated UTF-8, which means that a Unicode string cannot contain the U+0000 code point.

The value of a property is immutable: A Trusted Application can only retrieve it and cannot modify it. The value is set and controlled by the Implementation and MUST be trustable by the Trusted Applications.

The following ***Property Sets*** are exposed in the API:

- Each Trusted Application can access its own configuration properties. Some of these parameters affect the behavior of the TEE Implementation itself. Others can be used to configure the behavior of the TAs that this TA connects to.
- A TA instance can access a set of properties for each of its Clients. When the Client is a Trusted Application, the property set contains the configuration properties of that Trusted Application. Otherwise, it contains properties set by the Rich Execution Environment.
- Finally, a TA can access properties describing characteristics of the TEE Implementation itself.

Property names are case-sensitive and have a hierarchical structure with levels in the hierarchy separated by the dot character "`.`". Property names should use the reverse domain name convention to minimize the risk of collisions between properties defined by different organization, although this cannot really be enforced by an Implementation. For example, the ACME company should use the "`com.acme.`" prefix and properties standardized at ISO will use the "`org.iso.`" namespace.

This specification reserves the "`gpd.`" namespace and defines the meaning of a few properties in this namespace. Any Implementation MUST refuse to define properties in this namespace unless they meet this specification.

2.5 Trusted Storage API for Data and Keys

This specification defines an API that defines Trusted Storage for keys or general-purpose data. This API provides access to the following facilities:

- Trusted Storage for general-purpose data and key material with guarantees on the confidentiality and integrity of the data stored and atomicity of the operations that modify the storage
  - The Trusted Storage may be backed by non-secure resources as long as suitable cryptographic protection is applied, which must be as strong as the means used to protect the TEE code and data itself.
  - The Trusted Storage must be bound to a particular device, which means that it must be accessible or modifiable only by authorized TAs running in the same TEE and on the same device as when the data was created.
  - See the TEE System Architecture [2] §2.2, for more details on the security requirements for the Trusted Storage.
- Ability to hide sensitive key material from the TA itself
- Association of data and key: Any key object can be associated with a data stream and pure data objects contain only the data stream and no key material.
- Separation of storage among different TAs:
  - Each TA has access to its own storage space that is shared among all the instances of that TA but separated from the other TAs.

2.6 Cryptographic Operations API

This specification defines an API that provides the following cryptographic facilities:

- Generation and derivation of keys and key pairs
- Support for the following types of cryptographic algorithms:
  - Digests
  - Symmetric Ciphers
  - Message Authentication Codes (MAC)
  - Authenticated Encryption algorithms such as AES-CCM and AES-GCM
  - Asymmetric Encryption and Signature
  - Key Exchange algorithms
- Pre-allocation of cryptographic operations and key containers so that resources can be allocated ahead of time and reused for multiple operations and with multiple keys over time

2.7 Time API

This specification defines an API to access three sources of time:

- The ***System Time*** has an arbitrary non-persistent origin. It may use a secure dedicated hardware timer or be based on the REE timers.
- The ***TA Persistent Time*** is real-time and persistent but its origin is individually controlled by each TA. This allows each TA to independently synchronize its time with the external source of trusted time of its choice. The TEE itself is not required to have a defined trusted source of time.
- The ***REE Time*** is real-time but should not be more trusted than the REE and the user.

The level of trust that a Trusted Application can put in System Time and its TA Persistent Time is implementation-defined as a given Implementation may not include fully trustable hardware sources of time and hence may have to rely on untrusted real-time clocks and timers managed by the Rich Execution Environment. However, when a more trustable source of time is available, it is expected that it will be exposed to Trusted Applications through this Time API. Note that a Trusted Application can programmatically determine the level of protection of time sources by querying implementation properties (`gpd.tee.systemTime.protectionLevel` and `gpd.tee.TAPersistentTime.protectionLevel`).

2.8 Arithmetical API

The TEE Arithmetical API is a low-level API that complements the Cryptographic API when a Trusted Application must implement asymmetric algorithms, modes, or paddings not supported by the Cryptographic API.

The API provides arithmetical functions to work on big numbers and prime field elements. It provides operations including regular arithmetic, modular arithmetic, primality test, and fast modular multiplication that can be based on the Montgomery reduction or a similar technique.

3 Common Definitions

This chapter specifies the header file, common data types, constants, and parameter annotations used throughout the specification.

3.1 Header File

The header file for the TEE Internal API must have the name "`tee_internal_api.h`".

```
#include "tee_internal_api.h"
```

3.2 Data Types

3.2.1 Basic Types

This specification makes use of the integer and Boolean C types as defined in the C99 standard (ISO/IEC 9899:1999) [3]. The following basic types are used:

- `uint32_t`: Unsigned 32-bit integer
- `int32_t`: Signed 32-bit integer
- `uint16_t`: Unsigned 16-bit integer
- `int16_t`: Signed 16-bit integer
- `uint8_t`: Unsigned 8-bit integer
- `int8_t`: Signed 8-bit integer
- `bool`: Boolean type with the values `true` and `false`
- `char`: Character; used to denote a byte in a zero-terminated string encoded in UTF-8

In this specification, bits in integers are numbered from 0 (least-significant bit) to 7, 15, or 31 (most-significant bit), depending on the size of the integer.

3.2.2 TEE_Result, TEEC_Result

```
typedef uint32_t TEE_Result;
```

`TEE_Result` is the type used for return codes from the APIs.

For compatibility with the TEE Client API [1], the following alias of this type is also defined:

```
typedef TEE_Result TEEC_Result;
```

3.2.3 TEE_UUID, TEEC_UUID

```
typedef struct
{
   uint32_t timeLow;
   uint16_t timeMid;
   uint16_t timeHiAndVersion;
   uint8_t  clockSeqAndNode[8];
}
TEE_UUID;
```

`TEE_UUID` is the Universally Unique Resource Identifier type as defined in [4]. This type is used to identify Trusted Applications and clients.

UUIDs can be directly hard-coded in the Trusted Application code. For example, the UUID `79B77788-9789-4a7a-A2BE-B60155EEF5F3` can be hard-coded using the following code:

```
static const TEE_UUID myUUID =
{
    0x79b77788, 0x9789, 0x4a7a,
    { 0xa2, 0xbe, 0xb6, 0x1, 0x55, 0xee, 0xf5, 0xf3 }
};
```

For compatibility with the TEE Client API [1], the following alias of this type is also defined:

```
typedef TEE_UUID TEEC_UUID;
```

3.3 Constants

3.3.1 Error Codes

The following error codes are used throughout the APIs.

Table 3-1: API Error Codes

Constant Names and Aliases		Value
TEE_SUCCESS	TEEC_SUCCESS	0x00000000
TEE_ERROR_GENERIC	TEEC_ERROR_GENERIC	0xFFFF0000
TEE_ERROR_ACCESS_DENIED	TEEC_ERROR_ACCESS_DENIED	0xFFFF0001
TEE_ERROR_CANCEL	TEEC_ERROR_CANCEL	0xFFFF0002
TEE_ERROR_ACCESS_CONFLICT	TEEC_ERROR_ACCESS_CONFLICT	0xFFFF0003
TEE_ERROR_EXCESS_DATA	TEEC_ERROR_EXCESS_DATA	0xFFFF0004
TEE_ERROR_BAD_FORMAT	TEEC_ERROR_BAD_FORMAT	0xFFFF0005
TEE_ERROR_BAD_PARAMETERS	TEEC_ERROR_BAD_PARAMETERS	0xFFFF0006
TEE_ERROR_BAD_STATE	TEEC_ERROR_BAD_STATE	0xFFFF0007
TEE_ERROR_ITEM_NOT_FOUND	TEEC_ERROR_ITEM_NOT_FOUND	0xFFFF0008
TEE_ERROR_NOT_IMPLEMENTED	TEEC_ERROR_NOT_IMPLEMENTED	0xFFFF0009
TEE_ERROR_NOT_SUPPORTED	TEEC_ERROR_NOT_SUPPORTED	0xFFFF000A
TEE_ERROR_NO_DATA	TEEC_ERROR_NO_DATA	0xFFFF000B
TEE_ERROR_OUT_OF_MEMORY	TEEC_ERROR_OUT_OF_MEMORY	0xFFFF000C
TEE_ERROR_BUSY	TEEC_ERROR_BUSY	0xFFFF000D
TEE_ERROR_COMMUNICATION	TEEC_ERROR_COMMUNICATION	0xFFFF000E
TEE_ERROR_SECURITY	TEEC_ERROR_SECURITY	0xFFFF000F
TEE_ERROR_SHORT_BUFFER	TEEC_ERROR_SHORT_BUFFER	0xFFFF0010
TEE_PENDING		0xFFFF2000
TEE_ERROR_TIMEOUT		0xFFFF3001
TEE_ERROR_OVERFLOW		0xFFFF300F
TEE_ERROR_TARGET_DEAD	TEEC_ERROR_TARGET_DEAD	0xFFFF3024
TEE_ERROR_STORAGE_NO_SPACE		0xFFFF3041
TEE_ERROR_MAC_INVALID		0xFFFF3071
TEE_ERROR_SIGNATURE_INVALID		0xFFFF3072
TEE_ERROR_TIME_NOT_SET		0xFFFF5000
TEE_ERROR_TIME_NEEDS_RESET		0xFFFF5001

3.4 Parameter Annotations

This specification uses a set of patterns on the function parameters. Instead of repeating this pattern again on each occurrence, these patterns are referred to with ***Parameter Annotations***. It is expected that this will also help with systematically translating the APIs into languages other than the C language.

The following sub-sections list all the parameter annotations used in the specification.

Note that these annotations cannot be expressed in the C language. However, the *[in]*, *[inbuf]*, *[instring]*, *[instringopt]*, and *[ctx]* annotations can make use of the `const` C keyword. This keyword is omitted in the specification of the functions to avoid mixing the formal annotations and a less expressive C keyword. However, the C header file of a compliant Implementation SHOULD use the `const` keyword when these annotations appear.

3.4.1 [in], [out], and [inout]

The annotation *[in]* applies to a parameter that has a pointer type on a structure, a base type, or more generally a buffer of a size known in the context of the API call. If the size needs to be clarified, the syntax *[in(size)]* is used.

When this annotation is present on a parameter, it means that the API Implementation uses the pointer only for reading and does not accept shared memory.

When a Trusted Application calls an API function that contains a parameter annotated with *[in]*, the parameter MUST be entirely readable by the Trusted Application and MUST be entirely owned by the calling Trusted Application instance, as defined in section 4.11.1. In particular, this means that the parameter MUST NOT reside in a block of shared memory owned by a client of the Trusted Application. The Implementation MUST check these conditions and if they are not satisfied, the API call MUST panic the calling Trusted Application instance.

The annotation *[out]* and *[inout]* are equivalent to *[in]* but for write access and read-and-write access respectively.

Note that, as described in section 4.11.1, the `NULL` pointer MUST never be accessible to a Trusted Application. This means that a Trusted Application MUST NOT pass the `NULL` pointer in an *[in]* parameter, except perhaps if the buffer size is zero.

See the function `TEE_CheckMemoryAccessRights` in section 4.11.1 for more details about shared memory and the `NULL` pointer. See the function `TEE_Panic` in section 4.8.1 for information about Panics.

3.4.2 [outopt]

The *[outopt]* annotation is equivalent to *[out]* except that the caller can set the parameter to `NULL`, in which case the result MUST be discarded.

3.4.3 [inbuf]

The *[inbuf]* annotation applies to a pair of parameters of type `void*` and `size_t`. It means that the parameters describe an input data buffer. The entire buffer MUST be readable by the Trusted Application and there is no restriction on the owner of the buffer: It can reside in shared memory or in private memory.

The Implementation MUST check that the buffer is entirely readable and MUST panic the calling Trusted Application instance if that is not the case.

Because the `NULL` pointer is never readable, a Trusted Application cannot pass `NULL` in the first `void*` parameter unless the second `size_t` parameter is set to 0.

3.4.4 [outbuf]

The *[outbuf]* annotation applies to a pair of parameters of type `void*` and `size_t*`, herein referenced with the names `buffer` and `size`. It is used by API functions to return an output data buffer. The data buffer must be allocated by the calling Trusted Application and passed in the `buffer` parameter. Because the size of the output buffer cannot generally be determined in advance, the following convention is used:

- On entry, `*size` contains the number of bytes actually allocated in `buffer`. The buffer with this number of bytes MUST be entirely writable by the Trusted Application, otherwise the Implementation MUST panic the calling Trusted Application instance. In any case, the implementation MUST NOT write beyond this limit.
- On exit:
  - If the output fits in the output buffer, then the Implementation MUST write the output in `buffer` and MUST update `*size` with the actual size of the output in bytes.
  - If the output does not fit in the output buffer, then the implementation MUST update `*size` with the required number of bytes and MUST return `TEE_ERROR_SHORT_BUFFER`. It is implementation-dependent whether the output buffer is left untouched or contains part of the output. In any case, the TA should consider that its content is undefined after the function returns.

When the function returns `TEE_ERROR_SHORT_BUFFER`, it MUST NOT have performed the actual requested operation. It MUST just return the size of the output data.

Note that if the caller sets `*size` to 0, the function will always return `TEE_ERROR_SHORT_BUFFER` unless the actual output data is empty. In this case, the parameter `buffer` can take any value, e.g., `NULL`, as it will not be accessed by the Implementation. If `*size` is set to a non-zero value on entry, then `buffer` cannot be `NULL` because the buffer starting from the `NULL` address is never writable.

There is no restriction on the owner of the buffer: It can reside in shared memory or in private memory.

The parameter `size` must be considered as *[inout]*. That is, `size` MUST be readable and writable by the Trusted Application. The parameter `size` MUST NOT be `NULL` and MUST NOT reside in shared memory. The Implementation MUST check these conditions and panic the calling Trusted Application instance if they are not satisfied.

3.4.5 [outbufopt]

The *[outbufopt]* annotation is equivalent to *[outbuf]* but if the parameter `size` is set to `NULL`, then the function must behave as if the output buffer was not large enough to hold the entire output data and the output data MUST be discarded. In this case, the parameter `buffer` is ignored, but should normally be set to `NULL`, too.

Note the difference between passing a `size` pointer set to `NULL` and passing a `size` that points to 0. Assuming the function does not fail for any other reasons:

- If `size` is set to `NULL`, the function performs the operation, returns `TEE_SUCCESS`, and the output data is discarded.
- If `size` points to 0, the function does not perform the operation. It just updates `*size` with the output size and returns `TEE_ERROR_SHORT_BUFFER`.

3.4.6 [instring] and [instringopt]

The *[instring]* annotation applies to a single *[in]* parameter, which must contain a zero-terminated string of `char` characters. Because the buffer is *[in]*, it cannot reside in shared memory.

The *[instringopt]* annotation is equivalent to *[instring]* but the parameter can be set to `NULL` to denote the absence of a string.

3.4.7 [outstring] and [outstringopt]

The *[outstring]* annotation is equivalent to *[outbuf]*, but the output data is specifically a zero-terminated string of `char` characters. The size of the buffer must account for the zero terminator. The buffer may reside in shared memory.

The *[outstringopt]* annotation is equivalent to *[outstring]* but with *[outbufopt]* instead of *[outbuf]*, which means that `size` can be set to `NULL` to discard the output.

3.4.8 [ctx]

The *[ctx]* annotation applies to a `void*` parameter. It means that the parameter is not accessed by the Implementation, but will merely be stored to be provided to the Trusted Application later. Although a Trusted Application typically uses such parameters to store pointers to allocated structures, they can contain any value.

4 Trusted Core Framework API

This chapter defines the Trusted Core Framework API, defining OS-like APIs and infrastructure. It contains the following sections:

- Section 4.1, Data Types
- Section 4.2, Constants

  Common definitions used throughout the chapter.
- Section 4.3, TA Interface

  Defines the entry points that each TA must define.
- Section 4.4, Property Access Functions

  Defines the generic functions to access properties. These functions can be used to access TA Configuration Properties, Client Properties, and Implementation Properties.
- Section 4.5, Trusted Application Configuration Properties

  Defines the standard Trusted Application Configuration Properties.
- Section 4.6, Client Properties

  Defines the standard Client Properties.
- Section 4.7, Implementation Properties

  Defines the standard Implementation Properties.
- Section 4.8, Panics

  Defines the function `TEE_Panic`.
- Section 4.9, Internal Client API

  Defines the Internal Client API that allows a Trusted Application to act as a Client of another Trusted Application.
- Section 4.10, Cancellation

  Defines how a Trusted Application can handle client cancellation requests, acknowledge them, and mask or unmask the propagated effects of cancellation requests on cancellable functions.
- Section 4.11, Memory Management

  Defines how to check the access rights to memory buffers, how to access global variables, how to allocate memory (similar to malloc) and a few utility functions to fill or copy memory blocks.

4.1 Data Types

4.1.1 TEE_Identity

```
typedef struct
{
   uint32_t   login;
   TEE_UUID   uuid;
} TEE_Identity;
```

The TEE_Identity structure defines the full identity of a Client:

- login is one of the TEE_LOGIN_XXX constants. (See section 4.2.2.)
- uuid contains the client UUID or Nil (as defined in [4]) if not applicable.

4.1.2 TEE_Param

```
typedef union
{
   struct
   {
      void*    buffer; size_t   size;
   } memref;
   struct
   {
      uint32_t a, b;
   } value;
} TEE_Param;
```

This union describes one parameter passed by the Trusted Core Framework to the entry points TA_OpenSessionEntryPoint or TA_InvokeCommandEntryPoint or by the TA to the functions TEE_OpenTASession or TEE_InvokeTACommand.

Which of the field value or memref to select is determined by the parameter type specified in the argument paramTypes passed to the entry point. See section 4.3.6.1 and section 4.9.4 for more details on how this type is used.

4.1.3 TEE_TASessionHandle

```
typedef struct __TEE_TASessionHandle* TEE_TASessionHandle
```

TEE_TASessionHandle is an opaque handle on a TA Session. These handles are returned by the function TEE_OpenTASession specified in section 4.9.1.

4.1.4 TEE_PropSetHandle

```
typedef struct __TEE_PropSetHandle* TEE_PropSetHandle
```

TEE_PropSetHandle is an opaque handle on a property set or enumerator. These handles either are returned by the function TEE_AllocatePropertyEnumerator specified in section 4.4.7 or are one of the pseudo-handles defined in section 4.2.4.

4.2 Constants

4.2.1 Parameter Types

Table 4-1: Parameter Type Constants

Constant Name	Constant Value
TEE_PARAM_TYPE_NONE	0
TEE_PARAM_TYPE_VALUE_INPUT	1
TEE_PARAM_TYPE_VALUE_OUTPUT	2
TEE_PARAM_TYPE_VALUE_INOUT	3
TEE_PARAM_TYPE_MEMREF_INPUT	5
TEE_PARAM_TYPE_MEMREF_OUTPUT	6
TEE_PARAM_TYPE_MEMREF_INOUT	7

4.2.2 Login Types

Table 4-2: Login Type Constants

Constant Name	Constant Value
TEE_LOGIN_PUBLIC	0x00000000
TEE_LOGIN_USER	0x00000001
TEE_LOGIN_GROUP	0x00000002
TEE_LOGIN_APPLICATION	0x00000004
TEE_LOGIN_APPLICATION_USER	0x00000005
TEE_LOGIN_APPLICATION_GROUP	0x00000006
TEE_LOGIN_TRUSTED_APP	0xF0000000

4.2.3 Origin Codes

Table 4-3: Origin Code Constants

Constant Name	Constant Value
TEE_ORIGIN_API	0x00000001
TEE_ORIGIN_COMMS	0x00000002
TEE_ORIGIN_TEE	0x00000003
TEE_ORIGIN_TRUSTED_APP	0x00000004

4.2.4 Property Set Pseudo-Handles

Table 4-4: Property Set Pseudo-Handle Constants

Constant Name	Constant Value
TEE_PROPSET_CURRENT_TA	(TEE_PropSetHandle)0xFFFFFFFF
TEE_PROPSET_CURRENT_CLIENT	(TEE_PropSetHandle)0xFFFFFFFE
TEE_PROPSET_TEE_IMPLEMENTATION	(TEE_PropSetHandle)0xFFFFFFFD

4.2.5 Memory Access Rights

Table 4-5: Memory Access Rights Constants

Constant Name	Constant Value
TEE_ACCESS_READ	0x00000001
TEE_ACCESS_WRITE	0x00000002
TEE_ACCESS_ANY_OWNER	0x00000004

4.3 TA Interface

Each Trusted Application must provide the Implementation with a number of functions, collectively called the "TA interface". These functions are the entry points called by the Trusted Core Framework to create the instance, notify the instance that a new client is connecting, notify the instance when the client invokes a command, etc. These entry points cannot be registered dynamically by the Trusted Application code: They must be bound to the framework before the Trusted Application code is started.

Table 4-6 lists the functions in the TA interface.

Table 4-6: TA Interface Functions

TA Interface Function (Entry Point)	Description
`TA_CreateEntryPoint`	This is the Trusted Application constructor. It is called once and only once in the life-time of the Trusted Application instance. If this function fails, the instance is not created.
`TA_DestroyEntryPoint`	This is the Trusted Application destructor. The Trusted Core Framework calls this function just before the Trusted Application instance is terminated. The Framework MUST guarantee that no sessions are open when this function is called. When `TA_DestroyEntryPoint` returns, the Framework MUST collect all resources claimed by the Trusted Application instance.
`TA_OpenSessionEntryPoint`	This function is called whenever a client attempts to connect to the Trusted Application instance to open a new session. If this function returns an error, the connection is rejected and no new session is opened. In this function, the Trusted Application can attach an opaque `void*` context to the session. This context is recalled in all subsequent TA calls within the session.
`TA_CloseSessionEntryPoint`	This function is called when the client closes a session and disconnects from the Trusted Application instance. The Implementation guarantees that there are no active commands in the session being closed. The session context reference is given back to the Trusted Application by the Framework. It is the responsibility of the Trusted Application to deallocate the session context if memory has been allocated for it.
`TA_InvokeCommandEntryPoint`	This function is called whenever a client invokes a Trusted Application command. The Framework gives back the session context reference to the Trusted Application in this function call.

Table 4-7 summarizes client operations and the resulting Trusted Application effect.

Table 4-7: Effect of Client Operation on TA Interface

Client Operation	Trusted Application Effect

`TEEC_OpenSession`
or
`TEE_OpenTASession`
	If a new Trusted Application instance is needed to handle the session, `TA_CreateEntryPoint` is called.
	Then, `TA_OpenSessionEntryPoint` is called.

`TEEC_InvokeCommand`
or
`TEE_InvokeTACommand`
	`TA_InvokeCommandEntryPoint` is called.

`TEEC_CloseSession`
or
`TEE_CloseTASession`
	`TA_CloseSessionEntryPoint` is called.
	For a multi-instance TA or for a single-instance, non keep-alive TA, if the session closed was the last session on the instance, then `TA_DestroyEntryPoint` is called. Otherwise, the instance is kept until the TEE shuts down.

`TEEC_RequestCancellation` *or* The function `TEE_OpenTASession` or `TEE_InvokeTACommand` is cancelled or times out.	See section 4.10 for details on the effect of cancellation requests.
Client terminates unexpectedly	From the point of view of the TA instance, the behavior MUST be identical to the situation where the client does not terminate unexpectedly but, for all opened sessions: • requests the cancellation of all pending operations in that session, • waits for the completion of all these operations in that session, • and finally closes that session. Note that there is no way for the TA to distinguish between the client gracefully cancelling all its operations and closing all its sessions and the Implementation taking over when the client dies unexpectedly.

Interface Operation Parameters

When a Client opens a session on a Trusted Application or invokes a command, it can send *Operation Parameters* to the Trusted Application. The parameters encode the data associated with the operation. Up to four parameters can be sent in an operation. If these are insufficient, then one of the parameters may be used to carry further parameter data via a Memory Reference.

Each parameter can be individually typed by the Client as a "*Value Parameter*", carrying two 32-bit integers, or a "*Memory Reference Parameter*", carrying a pointer to a client-owned memory buffer. Each parameter is also tagged with a direction of data flow (input, output, or both input and output). For output Memory References, there is a built-in mechanism for the Trusted Applications to report the necessary size of the buffer in case of a too-short buffer. See section 4.3.6 for more information about the handling of parameters in the TA interface.

Note that Memory Reference Parameters typically point to memory owned by the client and shared with the Trusted Application for the duration of the operation. This is especially useful in the case of REE Clients to minimize the number of memory copies and the data footprint in case a Trusted Application must deal with large data buffers, for example to process a multimedia stream protected by DRM.

Security Considerations

The fact that Memory References may use memory directly shared with the client implies that the Trusted Application must be especially careful when handling such data: Even if the client is not allowed to access the shared memory buffer during an operation on this buffer, the Trusted OS usually cannot enforce this restriction. A badly-designed or rogue client may well change the content of the shared memory buffer at any time, even between two consecutive memory accesses by the Trusted Application. This means that the Trusted Application should be carefully written to avoid any security problem if this happens. If values in the buffer are security critical, the Trusted Application should always read data only once from a shared buffer and then validate it. It must not assume that data written to the buffer can be read unchanged later on.

Error Handling

All TA interface functions except `TA_DestroyEntryPoint` and `TA_CloseSessionEntryPoint` return an error code of type `TEE_Result`. The behavior of the Framework when an entry point returns an error depends on the entry point called:

- If `TA_CreateEntryPoint` returns an error, the Trusted Application instance is not created.
- If `TA_OpenSessionEntryPoint` returns an error code, the client connection is rejected. Additionally, the error code is propagated to the client as described below.
- If `TA_InvokeCommandEntryPoint` returns an error code, this error code is propagated to the client.
- `TA_CloseSessionEntryPoint` and `TA_DestroyEntryPoint` cannot return an error.

`TA_OpenSessionEntryPoint` and `TA_InvokeCommandEntryPoint` error codes are propagated to the client via the TEE Client API [1] or the Internal Client API with the origin set to `TEEC_ORIGIN_TRUSTED_APP`.

Client Properties

When a Client connects to a Trusted Application, the Framework associates the session with Client Properties. Trusted Applications can retrieve the identity and properties of their client by calling one of the property access functions with the `TEE_PROPSET_CURRENT_CLIENT`. The standard client properties are fully specified in section 4.6.

The TA_EXPORT keyword

Depending on the compiler used and the targeted platform, a TA entry point may need to be decorated with an annotation such as `__declspec(dllexport)` or similar. This annotation must be defined in the TEE Internal API header file as `TA_EXPORT` and placed between the entry point return type and function name as shown in the specification of each entry point.

4.3.1 TA_CreateEntryPoint

```
TEE_Result TA_EXPORT TA_CreateEntryPoint( void )
```

Description

The function TA_CreateEntryPoint is the Trusted Application's constructor, which the Framework calls when it creates a new instance of the Trusted Application.

To register instance data, the implementation of this constructor can use either global variables or the function TEE_SetInstanceData (described in section 4.11.2).

Return Value

- TEE_SUCCESS: If the instance is successfully created, the function must return TEE_SUCCESS.
- Any other value: If any other code is returned, then the instance is not created, and no other entry points of this instance will be called. The Framework MUST reclaim all resources and dereference all objects related to the creation of the instance.

  If this entry point was called as a result of a client opening a session, the error code is returned to the client and the session is not opened.

4.3.2 TA_DestroyEntryPoint

```
void TA_EXPORT TA_DestroyEntryPoint( void )
```

Description

The function TA_DestroyEntryPoint is the Trusted Application's destructor, which the Framework calls when the instance is being destroyed.

When the function TA_DestroyEntryPoint is called, the Framework guarantees that no client session is currently open. Once the call to TA_DestroyEntryPoint has been completed, no other entry point of this instance will ever be called.

Note that when this function is called, all resources opened by the instance are still available. It is only after the function returns that the Implementation MUST start automatically reclaiming resources left opened.

Return Value

This function can return no success or error code. After this function returns the Implementation MUST consider the instance destroyed and MUST reclaim all resources left open by the instance.

4.3.3 TA_OpenSessionEntryPoint

```
TEE_Result TA_EXPORT TA_OpenSessionEntryPoint(
                uint32_t   paramTypes,
       [inout] TEE_Param  params[4],
    [out][ctx] void**     sessionContext )
```

Description

The Framework calls the function `TA_OpenSessionEntryPoint` when a client requests to open a session with the Trusted Application. The open session request may result in a new Trusted Application instance being created as defined by the `gpd.ta.singleInstance` property described in section 4.5.

The client can specify parameters in an open operation which are passed to the Trusted Application instance in the arguments `paramTypes` and `params`. These arguments can also be used by the Trusted Application instance to transfer response data back to the client. See section 4.3.6 for a specification of how to handle the operation parameters.

If this function returns `TEE_SUCCESS`, the client is connected to a Trusted Application instance and can invoke Trusted Application commands. When the client disconnects, the Framework will eventually call the `TA_CloseSessionEntryPoint` entry point.

If the function returns any error, the Framework rejects the connection and returns the error code and the current content of the parameters to the client. The return origin is then set to `TEEC_ORIGIN_TRUSTED_APP`.

The Trusted Application instance can register a session data pointer by setting `*sessionContext`. The value of this pointer is not interpreted by the Framework, and is simply passed back to other `TA_` functions within this session. Note that `*sessionContext` may be set with a pointer to a memory allocated by the Trusted Application instance or with anything else, such as an integer, a handle, etc. The Framework will *not* automatically free `*sessionContext` when the session is closed; the Trusted Application instance is responsible for freeing memory if required.

During the call to `TA_OpenSessionEntryPoint` the client may request to cancel the operation. See section 4.10 for more details on cancellations. If the call to `TA_OpenSessionEntryPoint` returns `TEE_SUCCESS`, the client must consider the session as successfully opened and explicitly close it if necessary.

Parameters

- `paramTypes`: The types of the four parameters. See section 4.3.6.1 for more information.
- `params`: A pointer to an array of four parameters. See section 4.3.6.2 for more information.
- `sessionContext`: A pointer to a variable that can be filled by the Trusted Application instance with an opaque `void*` data pointer

Return Value

- `TEE_SUCCESS`: If the session is successfully opened
- Any other value: If the session could not be opened
  - The error code may be one of the pre-defined codes, or may be a new error code defined by the Trusted Application implementation itself. In any case, the Implementation MUST report the error code to the client with the origin `TEEC_ORIGIN_TRUSTED_APP`.

4.3.4 TA_CloseSessionEntryPoint

```
void TA_EXPORT TA_CloseSessionEntryPoint(
    [ctx] void*    sessionContext)
```

Description

The Framework calls the function `TA_CloseSessionEntryPoint` to close a client session.

The Trusted Application implementation is responsible for freeing any resources consumed by the session being closed. Note that the Trusted Application cannot refuse to close a session, but can hold the closing until it returns from `TA_CloseSessionEntryPoint`. This is why this function cannot return an error code.

Parameters

- `sessionContext`: The value of the `void*` opaque data pointer set by the Trusted Application in the function `TA_OpenSessionEntryPoint` for this session.

Return Value

This function can return no success or error code.

4.3.5 TA_InvokeCommandEntryPoint

```
TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(
      [ctx] void*       sessionContext,
            uint32_t    commandID,
            uint32_t    paramTypes,
    [inout] TEE_Param   params[4] )
```

Description

The Framework calls the function `TA_InvokeCommandEntryPoint` when the client invokes a command within the given session.

The Trusted Application can access the parameters sent by the client through the `paramTypes` and `params` arguments. It can also use these arguments to transfer response data back to the client. See section 4.3.6 for a specification of how to handle the operation parameters.

During the call to `TA_InvokeCommandEntryPoint` the client may request to cancel the operation. See section 4.10 for more details on cancellations.

A command is always invoked within the context of a client session. Thus, any session function (see section 4.6) can be called by the command implementation.

Parameter

- `sessionContext`: The value of the `void*` opaque data pointer set by the Trusted Application in the function `TA_OpenSessionEntryPoint`
- `commandID`: A Trusted Application-specific code that identifies the command to be invoked
- `paramTypes`: The types of the four parameters. See section 4.3.6.1 for more information.
- `params`: A pointer to an array of four parameters. See section 4.3.6.2 for more information.

Return Value

- `TEE_SUCCESS`: If the command is successfully executed, the function must return this value.
- Any other value: If the invocation of the command fails for any reason
  - The error code may be one of the pre-defined codes, or may be a new error code defined by the Trusted Application implementation itself. In any case, the Implementation MUST report the error code to the client with the origin `TEEC_ORIGIN_TRUSTED_APP`.

4.3.6 Operation Parameters in the TA Interface

When a client opens a session or invokes a command within a session, it can transmit operation parameters to the Trusted Application instance and receive response data back from the Trusted Application instance.

Arguments `paramTypes` and `params` are used to encode the operation parameters and their types which are passed to the Trusted Application instance. While executing the open session or invoke command entry points, the Trusted Application can also write in `params` to encode the response data.

4.3.6.1 Content of paramTypes Argument

The argument `paramTypes` encodes the type of each of the four parameters passed to an entry point. The content of `paramTypes` is implementation-dependent.

Each parameter type can take one of the TEE_PARAM_TYPE_XXX values listed in Table 4-1 on page 33. The type of each parameter determines whether the parameter is used or not, whether it is a Value or a Memory Reference, and the direction of data flow between the Client and the Trusted Application instance: Input (Client to Trusted Application instance), Output (Trusted Application instance to Client), or both Input and Output.

The following macros are available to decode `paramTypes`:

```
#define TEE_PARAM_TYPES(t0,t1,t2,t3) \
   ((t0) | ((t1) << 4) | ((t2) << 8) | ((t3) << 12))

#define TEE_PARAM_TYPE_GET(t, i) (((t) >> (i*4)) & 0xF)
```

The macro TEE_PARAM_TYPES can be used to construct a value that you can compare against an incoming `paramTypes` to check the type of all the parameters in one comparison, as in the following example:

```
if (paramTypes !=
            TEE_PARAM_TYPES(
                  TEE_PARAM_TYPE_MEMREF_INPUT,
                  TEE_PARAM_TYPE_MEMREF_OUPUT,
                  TEE_PARAM_TYPE_NONE,
                  TEE_PARAM_TYPE_NONE))
{
   /* Bad parameter types */
   return TEE_ERROR_BAD_PARAMETERS;
}
```

The macro TEE_PARAM_TYPE_GET can be used to extract the type of a given parameter from `paramTypes` if you need more fine-grained type checking.

4.3.6.2 Initial Content of params Argument

When the Framework calls the Trusted Application entry point, it initializes the content of `params[i]` as described in Table 4-8.

Table 4-8: Content of `params[i]` when Trusted Application Entry Point Is Called

Value of type[i]	Content of params[i] when the entry point is called
`TEE_PARAM_TYPE_NONE` `TEE_PARAM_TYPE_VALUE_OUTPUT`	Filled with zeros.
`TEE_PARAM_TYPE_VALUE_INPUT` `TEE_PARAM_TYPE_VALUE_INOUT`	`params[i].value.a` and `params[i].value.b` contain the two integers sent by the client

`TEE_PARAM_TYPE_MEMREF_INPUT`
`TEE_PARAM_TYPE_MEMREF_OUTPUT`
`TEE_PARAM_TYPE_MEMREF_INOUT`
	`params[i].memref.buffer` is a pointer to memory buffer shared by the client. This can be `NULL`.
	`params[i].memref.size` describes the size of the buffer. If `buffer` is `NULL`, `size` is guaranteed to be zero.

Note that if the Client is a Client Application that uses the TEE Client API [1], the Trusted Application cannot distinguish between a registered and a temporary Memory Reference. Both are encoded as one of the `TEE_PARAM_TYPE_MEMREF_XXX` types and a pointer to the data is passed to the Trusted Application.

Security Warning: *For a Memory Reference Parameter, the buffer may concurrently exist within the client and Trusted Application instance memory spaces. It must therefore be assumed that the client is able to make changes to the content of this buffer asynchronously at any moment. It is a security risk to assume otherwise.*

Any Trusted Application which implements functionality that needs some guarantee that the contents of a buffer are constant should copy the contents of a shared buffer into Trusted Application instance-owned memory.

To determine whether a given buffer is a Memory Reference or a buffer owned by the Trusted Application itself, the function `TEE_CheckMemoryAccessRights` *defined in section 4.11.1 can be used.*

4.3.6.3 Behavior of the Framework when the Trusted Application Returns

When the Trusted Application entry point returns, the Framework reads the content of each `params[i]` to determine what response data to send to the client, as described in Table 4-9.

Table 4-9: Interpretation of `params[i]` when Trusted Application Entry Point Returns

Value of type[i]	Behavior of the Framework when entry point returns

`TEE_PARAM_TYPE_NONE`
`TEE_PARAM_TYPE_VALUE_INPUT`
`TEE_PARAM_TYPE_MEMREF_INPUT`
	The content of `params[i]` is ignored.

`TEE_PARAM_TYPE_VALUE_OUTPUT` `TEE_PARAM_TYPE_VALUE_INOUT`	`params[i].value.a` and `params[i].value.b` contain the two integers sent to the client.
`TEE_PARAM_TYPE_MEMREF_OUTPUT` `TEE_PARAM_TYPE_MEMREF_INOUT`	The Framework reads `params[i].memref.size`: • If it is equal or less than the original value of `size`, it is considered as the actual size of the memory buffer. In this case, the Framework assumes that the Trusted Application has not written beyond this actual size and only this actual size will be synchronized with the client. • If it is greater than the original value of `size`, it is considered as a request for a larger buffer. In this case, the Framework assumes that the Trusted Application has not written anything in the buffer and no data will be synchronized.

4.3.6.4 Memory Reference and Memory Synchronization

Note that if a parameter is a Memory Reference, the memory buffer may be released or unmapped immediately after the operation completes. Also, some implementations may explicitly synchronize the contents of the memory buffer before the operation starts and after the operation completes.

As a consequence:

- The Trusted Application must not access the memory buffer after the operation completes. In particular, it cannot be used as a long-term communication means between the client and the Trusted Application instance. A Memory Reference must be accessed only during the lifetime of the operation.
- The Trusted Application must not attempt to write into a memory buffer of type `TEE_PARAM_TYPE_MEMREF_INPUT`.
  - It is a Programmer Error to attempt to do this but the Implementation is not required to detect this and the access may well be just ignored.
- For a Memory Reference Parameter marked as `OUTPUT` or `INOUT`, the Trusted Application can write in the entire range described by the initial content of `params[i].memref.size`. However, the Implementation must only guarantee that the client will observe the modifications below the final value of `size` and only if the final value is equal or less than the original value.

  For example, assume the original value of `size` is 100:
  - If the Trusted Application does not modify the value of `size`, the complete buffer is synchronized and the client is guaranteed to observe all the changes.
  - If the Trusted Application writes 50 in `size`, then the client is only guaranteed to observe the changes within the range from index 0 to index 49.
  - If the Trusted Application writes 200 in `size`, then no data is guaranteed to be synchronized with the client. However, the client will receive the new value of `size`. The Trusted Application can typically use this feature to tell the client that the Memory Reference was too small and request that the client retry with a Memory Reference of at least 200 bytes.

Failure to comply with these constraints will result in undefined behavior and is a Programmer Error.

4.4 Property Access Functions

This section defines a set of functions to access individual properties in a property set, to convert them into a variety of types (printable strings, integers, Booleans, binary blocks, etc.), and to enumerate the properties in a property set. These functions can be used to access TA Configuration Properties, ***Client Properties***, and Implementation Properties.

The property set is passed to each function in a pseudo-handle parameter. Table 4-10 lists the defined property sets.

Table 4-10: Property Sets

Pseudo-Handle	Meaning
`TEE_PROPSET_CURRENT_TA`	The configuration properties for the current Trusted Application. See section 4.5 for a definition of these properties.
`TEE_PROPSET_CURRENT_CLIENT`	The properties of the current client. This pseudo-handle is valid only in the context of the entry points `TA_OpenSessionEntryPoint`, `TA_InvokeCommandEntryPoint`, and `TA_CloseSessionEntryPoint`. See section 4.6 for a definition of these properties.
`TEE_PROPSET_TEE_IMPLEMENTATION`	The properties of the TEE Implementation itself. See section 4.7.

Properties can be retrieved and converted according to a variety of syntaxes:

- Printable strings encoded in UTF-8
- Binary block
- 32-bit unsigned integer
- Boolean
- UUID
- Identity (a pair composed of a login method and a UUID)

Implementations have much latitude on how they store properties internally. However, they MUST ensure consistency among the values returned by the property getters for the various types. For example, if a property can be successfully retrieved as an integer, then it must also retrievable as a printable string whose format must conform to the syntax for integers. This syntax is defined in the specification of the function `TEE_GetPropertyAsU32`.

In general, the consistency rules are defined in each of the `TEE_GetPropertyAsXXX` functions (described in the following sections).

Properties in a property set can also be enumerated. For this:

- Allocate a property enumerator using the function `TEE_AllocatePropertyEnumerator`.
- Start the enumeration by calling `TEE_StartPropertyEnumerator`, passing the pseudo-handle on the desired property set.
- Call the functions `TEE_GetProperty`[`AsXXX`] with the enumerator handle and a `NULL` name.

An enumerator provides the properties in an arbitrary order. In particular, they are not required to be sorted by name although a given implementation may ensure this.

4.4.1 TEE_GetPropertyAsString

```
TEE_Result TEE_GetPropertyAsString(
                TEE_PropSetHandle propsetOrEnumerator,
   [instringopt] char*             name,
   [outstring]   char*             valueBuffer, size_t* valueBufferLen )
```

Description

The TEE_GetPropertyAsString function performs a lookup in a property set to retrieve an individual property and convert its value into a printable string.

When the lookup succeeds, the implementation MUST convert the property into a printable string and copy the result into the buffer described by valueBuffer and valueBufferLen.

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and it must be encoded in UTF-8.
  - If hPropSet is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name cannot be NULL
- valueBuffer, valueBufferLen: Output buffer for the property value

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_SHORT_BUFFER: If the value buffer is not large enough to hold the whole property value

4.4.2 TEE_GetPropertyAsBool

```
TEE_Result TEE_GetPropertyAsBool(
                      TEE_PropSetHandle  propsetOrEnumerator,
        [instringopt] char*       name,
        [out]         bool*       value )
```

Description

The TEE_GetPropertyAsBool function retrieves a single property in a property set and converts its value to a Boolean.

If a property cannot be viewed as a Boolean, this function MUST return TEE_ERROR_BAD_FORMAT.

Otherwise, if this function succeeds, then calling the function TEE_GetPropertyAsString on the same name and with a sufficiently large output buffer MUST also succeed and return a string equal to "true" or "false" case-insensitive, depending on the value of the Boolean.

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and must be encoded in UTF-8.
  - If hPropSet is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name cannot be NULL.
- value: A pointer to the variable that will contain the value of the property on success or false on error.

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_BAD_FORMAT: If the property value cannot be converted to a Boolean

4.4.3 TEE_GetPropertyAsU32

```
TEE_Result TEE_GetPropertyAsU32(
                      TEE_PropSetHandle  propsetOrEnumerator,
        [instringopt] char*       name,
        [out]         uint32_t*   value )
```

Description

The TEE_GetPropertyAsU32 function retrieves a single property in a property set and converts its value to a 32-bit unsigned integer.

If a property cannot be viewed as a 32-bit unsigned integer, this function MUST return TEE_ERROR_BAD_FORMAT.

Otherwise, if this function succeeds, then calling the function TEE_GetPropertyAsString on the same name and with a sufficiently large output buffer MUST also succeed and return a string that is consistent with the following syntax:

```
integer:              decimal-integer
                    | hexadecimal-integer
                    | binary-integer

decimal-integer:      [0-9,_]+{K,M}?
hexadecimal-integer:  0[x,X][0-9,a-f,A-F,_]+
binary-integer:       0[b,B][0,1,_]+
```

Note that the syntax allows returning the integer either in decimal, hexadecimal, or binary format, that the representation can mix cases and can include underscores to separate groups of digits, and finally that the decimal representation may use 'K' or 'M' to denote multiplication by 1024 or 1048576 respectively.

For example, here are a few acceptable representations of the number 1024: "1K", "0X400", "0b100_0000_0000".

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and must be encoded in UTF-8.
  - If hPropSet is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name cannot be NULL.
- value: A pointer to the variable that will contain the value of the property on success, or zero on error.

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_BAD_FORMAT: If the property value cannot be converted to an unsigned 32-bit integer

4.4.4 TEE_GetPropertyAsBinaryBlock

```
TEE_Result TEE_GetPropertyAsBinaryBlock(
                      TEE_PropSetHandle  propsetOrEnumerator,
        [instringopt] char*       name,
        [outbuf]      void*       valueBuffer, size_t* valueBufferLen )
```

Description

The function TEE_GetPropertyAsBinaryBlock retrieves an individual property and converts its value into a binary block.

If a property cannot be viewed as a binary block, this function MUST return TEE_ERROR_BAD_FORMAT.

Otherwise, if this function succeeds, then calling the function TEE_GetPropertyAsString on the same name and with a sufficiently large output buffer MUST also succeed and return a string that is consistent with a Base64 encoding of the binary block as defined in RFC 2045 [6], section 6.8 but with the following tolerance:

- An Implementation is allowed not to encode the final padding '=' characters.
- An Implementation is allowed to insert characters that are not in the Base64 character set.

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and must be encoded in UTF-8.
  - If hPropSet is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name cannot be NULL.
- valueBuffer, valueBufferLen: Output buffer for the binary block

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_BAD_FORMAT: If the property cannot be retrieved as a binary block
- TEE_ERROR_SHORT_BUFFER: If the value buffer is not large enough to hold the whole property value

4.4.5 TEE_GetPropertyAsUUID

```
TEE_Result TEE_GetPropertyAsUUID(
                      TEE_PropSetHandle  propsetOrEnumerator,
        [instringopt] char*       name,
        [out]         TEE_UUID*   value )
```

Description

The function TEE_GetPropertyAsUUID retrieves an individual property and converts its value into a UUID.

If a property cannot be viewed as a UUID, this function MUST return TEE_ERROR_BAD_FORMAT.

Otherwise, if this function succeeds, then calling the function TEE_GetPropertyAsString on the same name and with a sufficiently large output buffer MUST also succeed and return a string that is consistent with the concrete syntax of UUIDs defined in RFC 4122. Note that this string may mix character cases.

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and must be encoded in UTF-8.
  - If hPropSet is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name cannot be NULL.
- value: A pointer filled with the UUID. Must not be NULL.

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_BAD_FORMAT: If the property cannot be converted into a UUID

4.4.6 TEE_GetPropertyAsIdentity

```
TEE_Result TEE_GetPropertyAsIdentity(
                        TEE_PropSetHandle    propsetOrEnumerator,
        [instringopt]   char*         name,
        [out]           TEE_Identity* value )
```

Description

The function TEE_GetPropertyAsIdentity retrieves an individual property and converts its value into a TEE_Identity.

If this function succeeds then retrieving the property as a printable string using TEE_GetPropertyAsString must return a string consistent with the following syntax:

```
identity: integer (':' uuid)?
```

where:

- The integer is consistent with the integer syntax described in the specification of the function TEE_GetPropertyAsU32 in section 4.4.3.
- If the identity UUID is Nil, then it can be omitted from the string representation of the property.

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and must be encoded in UTF-8.
  - If hPropSet is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name cannot be NULL.
- value: A pointer filled with the identity. Must not be NULL.

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_BAD_FORMAT: If the property value cannot be converted into an Identity

4.4.7 TEE_AllocatePropertyEnumerator

```
TEE_Result TEE_AllocatePropertyEnumerator(
      [out]      TEE_PropSetHandle*  enumerator )
```

Description

The function TEE_AllocatePropertyEnumerator allocates a property enumerator object. Once a handle on a property enumerator has been allocated, it can be used to enumerate properties in a property set using the function TEE_StartPropertyEnumerator.

Parameters

- enumerator: A pointer filled with an opaque handle on the property enumerator on success and with TEE_HANDLE_NULL on error

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_OUT_OF_MEMORY: If there are not enough resources to allocate the property enumerator

4.4.8 TEE_FreePropertyEnumerator

```
void TEE_FreePropertyEnumerator(
            TEE_PropSetHandle  enumerator )
```

Description

The function TEE_FreePropertyEnumerator deallocates a property enumerator object.

Parameters

- enumerator: A handle on the enumerator to free

4.4.9 TEE_StartPropertyEnumerator

```
void TEE_StartPropertyEnumerator(
            TEE_PropSetHandle  enumerator,
            TEE_PropSetHandle  propSet )
```

Description

The function TEE_StartPropertyEnumerator starts to enumerate the properties in an enumerator.

Once an enumerator is attached to a property set:

- Properties can be retrieved using one of the TEE_GetPropertyAsXXX functions, passing the enumerator handle as the property set and NULL as the name.
- The function TEE_GetPropertyName can be used to retrieve the name of the current property in the enumerator.
- The function TEE_GetNextProperty can be used to advance the enumeration to the next property in the property set.

Parameters

- enumerator: A handle on the enumerator
- propSet: A pseudo-handle on the property set to enumerate. Must be one of the TEE_PROPSET_XXX pseudo-handles.

4.4.10 TEE_ResetPropertyEnumerator

```
void TEE_ResetPropertyEnumerator(
            TEE_PropSetHandle  enumerator )
```

Description

The function TEE_ResetPropertyEnumerator resets a property enumerate to its state immediately after allocation. If an enumeration is currently started, it is abandoned.

Parameters

- enumerator: A handle on the enumerator to reset

4.4.11 TEE_GetPropertyName

```
TEE_Result TEE_GetPropertyName(
                TEE_PropSetHandle  enumerator,
   [outstring]  void*      nameBuffer, size_t* nameBufferLen )
```

Description

The function TEE_GetPropertyName gets the name of the current property in an enumerator.

The property name MUST be the valid UTF-8 encoding of a Unicode string containing no U+0000 code points.

Parameters

- enumerator: A handle on the enumerator
- nameBuffer, nameBufferLen: The buffer filled with the name

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If there is no current property either because the enumerator has not started or because it has reached the end of the property set
- TEE_ERROR_SHORT_BUFFER: If the name buffer is not large enough to contain the property name

4.4.12 TEE_GetNextProperty

```
TEE_Result TEE_GetNextProperty(
            TEE_PropSetHandle  enumerator)
```

Description

The function TEE_GetNextProperty advances the enumerator to the next property.

Parameters

- enumerator: A handle on the enumerator

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If the enumerator has reached the end of the property set or if it has not started

4.5 Trusted Application Configuration Properties

Each Trusted Application is associated with Configuration Properties that are accessible using the generic Property Access Functions and the TEE_PROPSET_CURRENT_TA pseudo-handle. This section defines a few standard configuration properties that affect the behavior of the Implementation. Other configuration properties can be defined:

- either by the Implementation to configure implementation-defined behaviors,
- or by the Trusted Application itself for its own configuration purposes.

The way properties are actually configured and attached to a Trusted Application is beyond the scope of the specification.

Table 4-11 defines the standard configuration properties for Trusted Applications.

Table 4-11: Trusted Application Standard Configuration Properties

Property Name	Type	Meaning
`gpd.ta.appID`	UUID	The identifier of the Trusted Application.
`gpd.ta.singleInstance`	Boolean	Whether the Implementation shall create a single TA instance for all the client sessions (if `true`) or shall create a separate instance for each client session (if `false`).
`gpd.ta.multiSession`	Boolean	Whether the Trusted Application instance supports multiple sessions. This property is ignored for multi-instance Trusted Applications.
`gpd.ta.instanceKeepAlive`	Boolean	Whether the Trusted Application instance context shall be preserved when there are no sessions connected to the instance. The instance context is defined as all writable data within the memory space of the Trusted Application instance, including the instance heap. This property is meaningful only when the `gpd.ta.singleInstance` is set to `true`. When this property is set to `false`, then the TA instance MUST be created when one or more sessions are opened on the TA and it MUST be destroyed when there are no more sessions opened on the instance. When this property is set to `true`, then the TA instance is terminated only when the TEE shuts down, which includes when the device goes through a system-wide global power cycle. Note that the TEE MUST NOT shut down whenever the REE does not shut down and keeps a restorable state, including when it goes through transitions into lower power states (hibernation, suspend, etc.). The exact moment when a keep-alive single instance is created is implementation-defined but it MUST be no later than the first session opening.

Property Name	Type	Meaning
`gpd.ta.dataSize`	Integer	Maximum estimated amount of dynamic data in bytes configured for the Trusted Application. The memory blocks allocated through `TEE_Malloc` are drawn from this space, as well as the task stacks. How this value precisely relates to the exact number and sizes of blocks that can be allocated is implementation-dependent.
`gpd.ta.stackSize`	Integer	Maximum stack size in bytes available to any task in the Trusted Application at any point in time. This corresponds to the stack size used by the TA code itself and does not include stack space possibly used by the Trusted Core Framework. For example, if this property is set to "`512`", then the Framework MUST guarantee that, at any time, the TA code can consume up to 512 bytes of stack and still be able to call any functions in the API.

4.6 Client Properties

This section defines the standard Client Properties, accessible using the generic Property Access Functions and the TEE_PROPSET_CURRENT_CLIENT pseudo-handle. Other non-standard client properties can be defined by specific implementations, but they must be defined outside the "gpd." namespace.

Note that Client Properties can be accessed only in the context of a TA entry point associated with a client, i.e., in one of the following entry point functions: TA_OpenSessionEntryPoint, TA_InvokeCommandEntryPoint, or TA_CloseSessionEntryPoint.

Table 4-12 defines the standard client properties.

Table 4-12: Standard Client Properties

Property Name	Type	Meaning
gpd.client.identity	Identity	Identity of the current client. This can be conveniently retrieved using the function TEE_GetPropertyAsIdentity (see section 4.4.6). A Trusted Application can use the client identity to perform access control. For example, it can refuse to open a session for a client that is not identified.

As shown in Table 4-13, the client identifier and the client properties that the Trusted Application can retrieve depend on the nature of the client and the method it has used to connect.

Table 4-13: Client Identities

Login Method	Meaning
TEE_LOGIN_PUBLIC	The client is in the Rich Execution Environment and is neither identified nor authenticated. The client has no identity and the UUID is the Nil UUID as defined in [4].
TEE_LOGIN_APPLICATION	The client application has been identified by the Rich Execution Environment independently of the identity of the user executing the application. The nature of this identification and the corresponding UUID is REE-specific.
TEE_LOGIN_USER	The client application has been identified by the Rich Execution Environment and the client UUID reflects the actual user that runs the calling application independently of the actual application.
TEE_LOGIN_GROUP	The client UUID reflects a group identity that is executing the calling application. The notion of group identity and the corresponding UUID is REE-specific.
TEE_LOGIN_APPLICATION_USER	The client UUID identifies both the calling application and the user that is executing it.
TEE_LOGIN_APPLICATION_GROUP	The client UUID identifies both the calling application and a group that is executing it.

Login Method	Meaning
TEE_LOGIN_TRUSTED_APP	The client is another Trusted Application. The client identity assigned to this session is the UUID of the calling Trusted Application. The client properties are all the configuration properties of the calling Trusted Application.
The range 0x80000000-0xEFFFFFFF is reserved for *implementation-defined* login methods.	The meaning of the Client UUID and the associated client properties are *implementation-defined*. If the Trusted Application does not support the particular implementation, it should assume that the client has minimum rights, i.e., rights equivalent to the login method TEE_LOGIN_PUBLIC.
The ranges 0x00000000-0x7FFFFFFF and 0xF0000000-0xFFFFFFFF are reserved for future standard login methods defined by GlobalPlatform.	

Client properties are meant to be managed by either the Rich OS or the Trusted OS and these must ensure that a Client cannot tamper with its own properties in the following sense:

- The property gpd.client.identity MUST always be determined by the Trusted OS and the determination of whether it is equal to TEE_LOGIN_TRUSTED_APP or not MUST be as trustworthy as the Trusted OS itself.
- When gpd.client.identity is equal to TEE_LOGIN_TRUSTED_APP then the Trusted OS MUST ensure that the remaining properties are equal to the properties of the calling TA up to the same level of trustworthiness that the target TA places in the Trusted OS.
- When gpd.client.identity is not equal to TEE_LOGIN_TRUSTED_APP, then the Rich OS is responsible for ensuring that the Client Application cannot tamper with its own properties.

Note that if a Client wants to transmit a property that is not synthesized by the Rich OS or Trusted OS, such as a password, then it must use a parameter to the session open operation or in subsequent commands.

4.7 Implementation Properties

The implementation properties can be retrieved by the generic Property Access Functions with the `TEE_PROPSET_IMPLEMENTATION` pseudo-handle.

Table 4-14 defines the standard implementation properties.

Table 4-14: Implementation Properties

Property Name	Type	Meaning
`gpd.tee.apiversion`	String	The version number of the API implementation. Its value for this version of the specification is the string "1.0".
`gpd.tee.description`	String	A description of the implementation. The content of this property is implementation-dependent but typically contains a version and build number of the implementation as well as other configuration information. Note that implementations are free to define their own non-standard identification property names, provided they are not in the "`gpd.`" namespace
`gpd.tee.deviceID`	UUID	A device identifier that must be globally unique among all GlobalPlatform TEEs whatever the manufacturer, vendor, or integration. **Implementer's Note** It is acceptable to derive this device identifier from statistically unique secret or public information, such as a Hardware Unique Key, die identifiers, etc. However, note that this property is intended to be public and exposed to any software running on the device, not only to Trusted Applications. The derivation must therefore be carefully designed so that it does not compromise secret information.
`gpd.tee.systemTime.protectionLevel`	Integer	The protection level provided by the system time implementation. See the function `TEE_GetSystemTime` in section 7.2.1 for more details.
`gpd.tee.TAPersistentTime.protectionLevel`	Integer	The protection level provided for the TA Persistent Time. See the function `TEE_GetTAPersistentTime` in section 7.2.3 for more details.

Property Name	Type	Meaning
`gpd.tee.arith.maxBigIntSize`	Integer	Maximum size in bits of the big integers for all the functions in the TEE Arithmetical API specified in Chapter 8. Beyond this limit, some of the functions MAY panic due to insufficient preallocated resources or hardware limitations.

4.8 Panics

4.8.1 TEE_Panic

```
void TEE_Panic(TEE_Result panicCode);
```

Description

The TEE_Panic function raises a Panic in the Trusted Application instance.

When a Trusted Application calls the TEE_Panic function, the current instance MUST be destroyed and all the resources opened by the instance MUST be reclaimed. All sessions opened from the panicking instance on another TA must be gracefully closed and all cryptographic objects and operations must be closed properly.

When an instance panics, its clients receive the error code TEE_ERROR_TARGET_DEAD of origin TEE_ORIGIN_TEE until they close their session. This applies to Rich Execution Environment clients calling through the TEE Client API [1] and to Trusted Execution Environment clients calling through the Internal Client API.

Once an instance is panicked, no TA entry point is ever called again for this instance, not even TA_DestroyEntryPoint. The caller cannot expect that the TEE_Panic function will return.

Parameters

- panicCode: An informative panic code defined by the TA. May be displayed in traces if traces are available.

4.9 Internal Client API

This API allows a Trusted Application to act as a client to another Trusted Application.

4.9.1 TEE_OpenTASession

```
TEE_Result TEE_OpenTASession(
      [in] TEE_UUID*             destination,
           uint32_t              cancellationRequestTimeout,
           uint32_t              paramTypes,
   [inout] TEE_Param             params[4],
     [out] TEE_TASessionHandle*  session,
     [out] uint32_t*             returnOrigin)
```

Description

The function `TEE_OpenTASession` opens a new session with a Trusted Application.

The destination Trusted Application is identified by its UUID passed in `destination`. This UUID can be hardcoded in the caller code.

An initial set of four parameters can be passed during the operation. See section 4.9.4 for a detailed specification of how these parameters are passed in the `paramTypes` and `params` arguments.

The result of this function is returned both in the return value and the return origin, stored in the variable pointed to by `returnOrigin`:

- If the return origin is different from `TEE_ORIGIN_TRUSTED_APP`, then the function has failed before it could reach the target Trusted Application. The possible error codes are listed in “Return Value” below.
- If the return origin is `TEE_ORIGIN_TRUSTED_APP`, then the meaning of the return value depends on the protocol exposed by the target Trusted Application. However, if `TEE_SUCCESS` is returned, it always means that the session was successfully opened and if the function returns a value different from `TEE_SUCCESS`, it means that the session opening failed.

When the session is successfully opened, i.e., when the function returns `TEE_SUCCESS`, a valid session handle is written into `*session`. Otherwise, the value `TEE_HANDLE_NULL` is written into `*session`.

When a session is to be closed, the client Trusted Application must call the function `TEE_CloseTASession` with the session handle.

Parameters

- `destination`: A pointer to a `TEE_UUID` structure containing the UUID of the destination Trusted Application
- `cancellationRequestTimeout`: Timeout in milliseconds or the special value `TEE_TIMEOUT_INFINITE` if there is no timeout. After the timeout expires, a cancellation request for the operation must be automatically sent.
- `paramTypes`: The types of all parameters passed in the operation. See section 4.9.4 for more details.
- `params`: The parameters passed in the operation. See section 4.9.4 for more details.
- `session`: A pointer to a variable that will receive the client session handle. The pointer must not be `NULL`. The value is set to `TEE_HANDLE_NULL` upon error.
- `returnOrigin`: A pointer to a variable which will contain the return origin. This field may be `NULL` if the return origin is not needed.

Return Value

If the return origin is different from `TEE_ORIGIN_TRUSTED_APP`, one of the following error codes can be returned:

- `TEE_ERROR_OUT_OF_MEMORY`: If not enough resources are available to open the session
- `TEE_ERROR_ITEM_NOT_FOUND`: If no Trusted Application matches the requested destination UUID
- `TEE_ERROR_ACCESS_DENIED`: If access to the destination Trusted Application is denied
- `TEE_ERROR_BUSY`: If the destination Trusted Application does not allow more than one session at a time and already has a session in progress
- `TEE_ERROR_TARGET_DEAD`: If the destination Trusted Application has panicked during the operation

If the return origin is `TEE_ORIGIN_TRUSTED_APP`, the return code is defined by the protocol exposed by the destination Trusted Application. In any case, a return code set to `TEE_SUCCESS` means that the session was successfully opened and a return code different from `TEE_SUCCESS` means that the opening failed.

4.9.2 TEE_CloseTASession

```
void TEE_CloseTASession(TEE_TASessionHandle session)
```

Description

The function `TEE_CloseTASession` closes a client session.

Parameters

- `session`: An opened session handle

4.9.3 TEE_InvokeTACommand

```
TEE_Result TEE_InvokeTACommand(
           TEE_TASessionHandle  session,
           uint32_t             cancellationRequestTimeout,
           uint32_t             commandID,
           uint32_t             paramTypes,
   [inout] TEE_Param            params[4],
     [out] uint32_t*            returnOrigin)
```

Description

The function TEE_InvokeTACommand invokes a command within a session opened between the client Trusted Application instance and a destination Trusted Application instance.

The parameter session must reference a valid session handle opened by TEE_OpenTASession.

Up to four parameters can be passed during the operation. See section 4.9.4 for a detailed specification of how these parameters are passed in the paramTypes and params arguments.

The result of this function is returned both in the return value and the return origin, stored in the variable pointed to by returnOrigin:

- If the return origin is different from TEE_ORIGIN_TRUSTED_APP, then the function has failed before it could reach the destination Trusted Application. The possible error codes are listed in “Return Value” below.
- If the return origin is TEE_ORIGIN_TRUSTED_APP, then the meaning of the return value is determined by the protocol exposed by the destination Trusted Application. It is recommended that the Trusted Application developer choose TEE_SUCCESS (0) to indicate success in their protocol, as this makes it possible to determine success or failure without looking at the return origin.

Parameters

- session: An opened session handle
- cancellationRequestTimeout: Timeout in milliseconds or the special value TEE_TIMEOUT_INFINITE if there is no timeout. After the timeout expires, a cancellation request for the operation must be automatically sent.
- commandID: The identifier of the Command to invoke. The meaning of each Command Identifier must be defined in the protocol exposed by the target Trusted Application.
- paramTypes: The types of all parameters passed in the operation. See section 4.9.4 for more details.
- params: The parameters passed in the operation. See section 4.9.4 for more details.
- returnOrigin: A pointer to a variable which will contain the return origin. This field may be NULL if the return origin is not needed.

Return Value

If the return origin is different from TEE_ORIGIN_TRUSTED_APP, one of the following error codes can be returned:

- TEE_ERROR_OUT_OF_MEMORY: If not enough resources are available to perform the operation
- TEE_ERROR_TARGET_DEAD: If the destination Trusted Application has panicked during the operation

If the return origin is TEE_ORIGIN_TRUSTED_APP, the return code is defined by the protocol exposed by the destination Trusted Application.

4.9.4 Operation Parameters in the Internal Client API

The functions TEE_OpenTASession and TEE_InvokeTACommand take paramTypes and params as arguments. The calling Trusted Application can use these arguments to pass up to four parameters.

Each of the parameters has a type, which is one of the TEE_PARAM_TYPE_XXX values listed in Table 4-1 on page 33. The content of paramTypes should be built using the macro TEE_PARAM_TYPES (see section 4.3.6.1).

Unless all parameter types are set to TEE_PARAM_TYPE_NONE, params must not be NULL and must point to an array of four TEE_Param elements. Each of the params[i] is interpreted as follows.

When the operation starts, the Framework reads the parameters as described in Table 4-15.

Table 4-15: Interpretation of params[i] on Entry to Internal Client API

Parameter Type	Interpretation of params[i]
TEE_PARAM_TYPE_NONE TEE_PARAM_TYPE_VALUE_OUTPUT	Ignored.
TEE_PARAM_TYPE_VALUE_INPUT TEE_PARAM_TYPE_VALUE_INOUT	Contains two integers in params[i].value.a and params[i].value.b.
TEE_PARAM_TYPE_MEMREF_INPUT TEE_PARAM_TYPE_MEMREF_OUTPUT TEE_PARAM_TYPE_MEMREF_INOUT	params[i].memref.buffer and params[i].memref.size must be initialized with a memory buffer that is accessible with the access rights described in the type. The buffer can be NULL, in which case size must be set to 0.

During the operation, the destination Trusted Application can update the contents of the OUTPUT or INOUT Memory References.

When the operation completes, the Framework updates the structure params[i] as described in Table 4-16.

Table 4-16: Effects of Internal Client API on params[i]

Parameter Type	Effects on params[i]
TEE_PARAM_TYPE_NONE TEE_PARAM_TYPE_VALUE_INPUT TEE_PARAM_TYPE_MEMREF_INPUT	Unchanged.
TEE_PARAM_TYPE_VALUE_OUTPUT TEE_PARAM_TYPE_VALUE_INOUT	params[i].value.a and params[i].value.b are updated with the value sent by the destination Trusted Application.
TEE_PARAM_TYPE_MEMREF_OUTPUT TEE_PARAM_TYPE_MEMREF_INOUT	params[i].memref.size is updated to reflect the actual or requested size of the buffer.

4.10 Cancellation Functions

This section defines functions for Trusted Applications to handle cancellation requested by a Client Application.

When a Client Application requests cancellation using the function `TEEC_RequestCancellation` of the TEE Client API [1], the implementation must do the following:

- If the operation has not reached the TA yet but has been queued in the TEE, then it MUST be retired from the queue and fail with the error code `TEEC_ERROR_CANCEL` and the origin `TEEC_ORIGIN_TEE`.
- If the operation has been transmitted to the Trusted Application, the implementation MUST set the ***Cancellation Flag*** of the task executing the command.
- If the Trusted Application has unmasked the effects of cancellation by using the function `TEE_UnmaskCancellation`, and if the task is engaged in a cancellable function when the Cancellation Flag is set, then that cancellable function is interrupted. The Trusted Application can detect that the function has been interrupted because it returns `TEE_ERROR_CANCEL`. It can then execute cleanup code and possibly fail the current client operation, although it may well report a success.
  - Note that this version of the specification defines a single cancellable function, which is the `TEE_Wait` function. Future versions may define other cancellable functions, in particular in the domain of user interactions.
  - The functions `TEE_OpenTASession` and `TEE_InvokeTACommand`, while not cancellable per se, must transmit cancellation requests: If the Cancellation Flag is set and the effects of cancellation are not masked, then the Trusted Core Framework MUST consider that the cancellation of the corresponding operation is requested.
- When the Cancellation Flag is set for a given task, the function `TEE_GetCancellationFlag` MUST return `true`, but only in the case the cancellations are not masked. This allows the Trusted Application to poll the Cancellation Flag, for example, when it is engaged in a lengthy active computation not using cancellable functions such as `TEE_Wait`.

4.10.1 TEE_GetCancellationFlag

```
bool TEE_GetCancellationFlag( void )
```

Description

The `TEE_GetCancellationFlag` function determines whether the current task's Cancellation Flag is set. If cancellations are masked, this function must return `false`.

Return Value

- `false` if the cancellation flag is not set or if cancellations are masked
- `true` if the cancellation flag is set and cancellations are not masked

4.10.2 TEE_UnmaskCancellation

```
bool TEE_UnmaskCancellation( void )
```

Description

The TEE_UnmaskCancellation function unmasks the effects of cancellation for the current task.

When cancellation requests are unmasked, the Cancellation Flag interrupts cancellable functions such as TEE_Wait and requests the cancellation of operations started with TEE_OpenTASession or TEE_InvokeTACommand.

By default, tasks created to handle a TA entry point have cancellation masked, so that a TA does not have to cope with the effects of cancellation requests.

Return Value

- true if cancellations were masked prior to calling this function
- false otherwise

4.10.3 TEE_MaskCancellation

```
bool TEE_MaskCancellation( void )
```

Description

The TEE_MaskCancellation function masks the effects of cancellation for the current task.

When cancellation requests are masked, the Cancellation Flag does not have an effect on the cancellable functions and cannot be retrieved using TEE_GetCancellationFlag.

By default, tasks created to handle a TA entry point have cancellation masked, so that a TA does not have to cope with the effects of cancellation requests.

Return Value

- true if cancellations were masked prior to calling this function
- false otherwise

4.11 Memory Management Functions

This section defines the following functions:

- A function to check the access rights of a given buffer. This can be used in particular to check if the buffer belongs to shared memory.
- Access to an instance data register, which provides a possibly more efficient alternative to using read-write C global variables
- A malloc facility
- A few utilities to copy and fill data blocks

4.11.1 TEE_CheckMemoryAccessRights

```
TEE_Result TEE_CheckMemoryAccessRights(
                    uint32_t accessFlags,
                    void*    buffer, size_t size)
```

Description

The TEE_CheckMemoryAccessRights function causes the Implementation to examine a buffer of memory specified in the parameters buffer and size and to determine whether the current Trusted Application instance has the access rights requested in the parameter accessFlags. If the characteristics of the buffer are compatible with accessFlags, then the function returns TEE_SUCCESS. Otherwise, it returns TEE_ERROR_ACCESS_DENIED. Note that the buffer should not be accessed by the function, but the Implementation should check the access rights based on the address of the buffer and internal memory management information.

The parameter accessFlags can contain one or more of the following flags:

- TEE_MEMORY_ACCESS_READ: Check that the buffer is entirely readable by the current Trusted Application instance.
- TEE_MEMORY_ACCESS_WRITE: Check that the buffer is entirely writable by the current Trusted Application instance.
- TEE_MEMORY_ACCESS_ANY_OWNER:
  - If this flag is *not* set, then the function checks that the buffer is not shared, i.e., whether it can be safely passed in an *[in]* or *[out]* parameter.
  - If this flag is set, then the function does not check ownership. It returns TEE_SUCCESS if the Trusted Application instance has read or write access to the buffer, independently of whether the buffer resides in memory owned by a Client or not.
- All other flags are reserved for future use and SHOULD be set to 0.

The result of this function is valid only until one of the following events occurs:

- The buffer is contained in a memory block previously allocated with TEE_Malloc or TEE_Realloc, and this block is deallocated with TEE_Free or reallocated with TEE_Realloc.
- One of the entry points of the Trusted Application returns. In this case, the Implementation is allowed to unmap or recycle the data buffers passed to the entry point in the TEE_Param structures.

In these two situations, the access rights of a given buffer MAY change and the Trusted Application SHOULD call the function TEE_CheckMemoryAccessRights again.

When this function returns TEE_SUCCESS, and as long as this result is still valid, the Implementation MUST guarantee the following properties:

- For the flag TEE_MEMORY_ACCESS_READ and TEE_MEMORY_ACCESS_WRITE, the Implementation MUST guarantee that subsequent read or write accesses by the Trusted Application wherever in the buffer will succeed and will not panic.
- When the flag TEE_MEMORY_ACCESS_ANY_OWNER is not set, the Implementation MUST guarantee that the memory buffer is owned either by the Trusted Application instance or by a more trusted component, and cannot be controlled, modified, or observed by a less trusted component, such as the Client of the Trusted Application. This means that the Trusted Application can assume the following guarantees:
  - **Read-after-read consistency**: If the Trusted Application performs two successive read accesses to the buffer at the same address and if, between the two read accesses, there is no write access, either direct or indirect through the API, then the two reads MUST return the same result.
  - **Read-after-write consistency**: If the Trusted Application writes some data in the buffer and subsequently reads the same address and if there is no write access in between, the read MUST return that data.
  - **Non-observability**: If the Trusted Application writes some data in the buffer, then the data MUST NOT be observable by components less trusted than the Trusted Application itself.

Note that when true memory sharing is implemented between Clients and the Trusted Application, the Memory Reference Parameters passed to the TA entry points will typically not satisfy these requirements. In this case, the function TEE_CheckMemoryAccessRights MUST return TEE_ERROR_ACCESS_DENIED. The code handling such buffers must be especially careful to avoid security issues brought by this lack of guarantees. For example, it can read each byte in the buffer only once and refrain from writing temporary data in the buffer.

Additionally, the Implementation MUST guarantee that some types of memory blocks have a minimum set of access rights:

- The following blocks MUST allow read and write accesses and MUST be owned by the Trusted Application instance:
  - All blocks returned by TEE_Malloc or TEE_Realloc
  - All the local and global non-const C variables
  - The TEE_Param structures passed to the entry points TA_OpenSessionEntryPoint and TA_InvokeCommandEntryPoint. This applies to the immediate contents of the TEE_Param structures, but not to the pointers contained in the fields of such structures, which can of course point to memory owned by the client. Note that this also means that these TEE_Param structures MUST NOT directly point to the corresponding structures in the TEE Client API [1] or the Internal Client API. The Implementation MUST perform a copy into a safe TA-owned memory buffer before passing the structures to the entry points.
- The following blocks MUST allow read accesses and MUST be owned by the Trusted Application instance:
  - The code of the Trusted Application itself
  - All const local or global C variables

- When a particular parameter passed in the structure `TEE_Param` to a TA entry point is a Memory Reference as specified in its parameter type, then this block, as described by the initial values of the fields `buffer` and `size` in that structure, MUST allow read and/or write accesses as specified in the parameter type. As noted above, this buffer is not required to reside in memory owned by the TA instance.

Finally, any Implementation MUST also guarantee that the `NULL` pointer cannot be dereferenced. If a Trusted Application attempts to read one byte at the address `NULL`, it MUST panic. This guarantee MUST extend to a segment of addresses starting at `NULL`, but the size of this segment is implementation-dependent.

Parameters

- `buffer`, `size`: The description of the buffer to check
- `accessFlags`: The access flags to check

Return Value

- `TEE_SUCCESS`: If the entire buffer allows the requested accesses
- `TEE_ERROR_ACCESS_DENIED`: If at least one byte in the buffer is not accessible with the requested accesses

Panic Reasons

This function MUST NOT panic for any reason.

4.11.2 TEE_SetInstanceData

```
void TEE_SetInstanceData(
           [ctx] void* instanceData )
```

Description

The TEE_SetInstanceData and TEE_GetInstanceData functions provide an alternative to writable global data (writable variables with global scope and writable static variables with global or function scope). While an Implementation MUST support C global variables, using these functions may be sometimes more efficient, especially if only a single instance data variable is required.

These two functions can be used to register and access an instance variable. Typically this instance variable can be used to hold a pointer to a Trusted Application-defined memory block containing any writable data that needs instance global scope, or writable static data that needs instance function scope.

An equivalent session context variable for managing session global and static data exists for sessions (see TA_OpenSessionEntryPoint, TA_InvokeCommandEntryPoint, and TA_CloseSessionEntryPoint in section 4.3).

This function sets the Trusted Application instance data pointer. The data pointer can then be retrieved by the Trusted Application instance by calling the TEE_GetInstanceData function.

Parameters

- instanceData: A pointer to the global Trusted Application instance data. This pointer may be NULL.

4.11.3 TEE_GetInstanceData

```
[ctx] void* TEE_GetInstanceData( void )
```

Description

The TEE_GetInstanceData function retrieves the instance data pointer set by the Trusted Application using the TEE_GetInstanceData function.

Return Value

The value returned is the previously set pointer to the Trusted Application instance data, or NULL if no instance data pointer has yet been set.

4.11.4 TEE_Malloc

```
void* TEE_Malloc( size_t size, uint32_t hint )
```

Description

The TEE_Malloc function allocates space for an object whose size in bytes is specified in the parameter size.

The pointer returned is guaranteed to be aligned such that it may be assigned as a pointer to any of the basic C types.

The parameter hint is a hint to the allocator. In this version of the specification, only one hint is defined. This parameter is nonetheless included so that the Trusted Applications may refer to various pools of memory or request special characteristics for the allocated memory by using an implementation-defined hint. Future versions of this specification may introduce additional standard hints.

The hint must be attached to the allocated block and should be used when the block is reallocated with TEE_Realloc.

If the space cannot be allocated, a NULL pointer is returned.

Parameter

- size: The size of the buffer to be allocated.
- hint: A hint to the allocator. Currently defined values are as follows:
  - The default value, 0, guarantees that the returned block of memory is filled with zeros.
  - Values in the range [0x00000001, 0x7FFFFFFF] are reserved for future version of this specification.
  - Values in the range [0x80000000, 0xFFFFFFFF] can be used for implementation-defined hints.

Return Value

Upon successful completion, with size not equal to zero, the function returns a pointer to the allocated space. If the space cannot be allocated, a NULL pointer is returned.

If the size of the requested space is zero, the value returned is undefined but guaranteed to be different from NULL and MUST NOT be accessed by the Trusted Application.

4.11.5 TEE_Realloc

```
void* TEE_Realloc(
           [in] void*    buffer,
                uint32_t newSize )
```

Description

The TEE_Realloc function changes the size of the memory object pointed to by buffer to the size specified by nNewSize.

The content of the object remains unchanged up to the lesser of the new and old sizes. Space in excess of the old size contains unspecified content.

If the new size of the memory object requires movement of the object, the space for the previous instantiation of the object is deallocated. If the space cannot be allocated, the original object remains allocated, and this function returns a NULL pointer.

If buffer is NULL, TEE_Realloc is equivalent to TEE_Malloc for the specified size.

It is a Programmer Error if buffer does not match a pointer previously returned by TEE_Malloc or TEE_Realloc, or if the space has previously been deallocated by a call to TEE_Free or TEE_Realloc.

If the hint initially provided when the block was allocated with TEE_Malloc is 0, then the extended space is filled with zeros. In general, the function TEE_Realloc SHOULD allocate the new memory buffer using exactly the same hint as for the buffer initially allocated with TEE_Malloc. In any case, it MUST NOT downgrade the security or performance characteristics of the buffer.

Note that any pointer returned by TEE_Malloc or TEE_Realloc and not yet freed or reallocated can be passed to TEE_Realloc. This includes the special non-NULL pointer returned when an allocation for 0 bytes is requested.

Parameters

- buffer: The pointer to the object to be reallocated
- newSize: The new size required for the object

Return Value

Upon successful completion, TEE_Realloc returns a pointer to the (possibly moved) allocated space.

If there is not enough available memory, TEE_Realloc returns a NULL pointer.

4.11.6 TEE_Free

```
void TEE_Free(void *buffer)
```

Description

The TEE_Free function causes the space pointed to by buffer to be deallocated; that is, made available for further allocation.

If buffer is a NULL pointer, TEE_Free does nothing. Otherwise, it is a Programmer Error if the argument does not match a pointer previously returned by the TEE_Malloc or TEE_Realloc, or if the space has been deallocated by a call to TEE_Free or TEE_Realloc.

Parameter

- buffer: The pointer to the memory block to be freed

4.11.7 TEE_MemMove

```
void  TEE_MemMove(
        [out] void*     dest,
         [in] void*     src,
              uint32_t  size );
```

Description

The TEE_MemMove function copies size bytes from the object pointed to by src into the object pointed to by dest.

Copying takes place as if the size bytes from the object pointed to by src are first copied into a temporary array of size bytes that does not overlap the objects pointed to by dest and src, and then the size bytes from the temporary array are copied into the object pointed to by dest.

Parameters

- dest: A pointer to the destination buffer
- src: A pointer to the source buffer
- size: The number of bytes to be copied

Note that the buffers dest and src can reside in any kinds of memory, including shared memory.

4.11.8 TEE_MemCompare

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

Description

The TEE_MemCompare function compares the first size bytes of the object pointed to by buffer1 to the first size bytes of the object pointed to by buffer2.

Parameters

- buffer1: A pointer to the first buffer
- buffer2: A pointer to the second buffer
- size: The number of bytes to be compared

Note that buffer1 and buffer2 can reside in any kinds of memory, including shared memory.

Return Value

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type uint8_t) that differ in the objects being compared.

- If the first byte that differs is higher in buffer1, then return an integer greater than zero.
- If the first size bytes of the two buffers are identical, then return zero.
- If the first byte that differs is higher in buffer2, then return an integer lower than zero.

4.11.9 TEE_MemFill

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

Description

The TEE_MemFill function writes the byte x (converted to a uint8_t) into the first size bytes of the object pointed to by buffer.

Parameters

- buffer: A pointer to the destination buffer
- x: The value to be set
- size: The number of bytes to be set

Note that buffer can reside in any kinds of memory, including shared memory.

5 Trusted Storage API for Data and Keys

This chapter includes the following sections:

5.1 Summary of Features and Design

This section provides a summary of the features and design of the Trusted Storage API.

- Each TA has access to a set of ***Trusted Storage Spaces***, identified by 32-bit ***Storage Identifiers***.
  - The current version of this specification defines a single Trusted Storage Space for each TA, which is its own private storage space. The objects in this storage space are accessible only to the TA that created them and are not visible to other TAs.
  - Other storage identifiers may be defined in future versions of this specification, e.g., to refer to storage spaces shared among multiple TAs or for communicating between boot-time entities and run-time Trusted Applications.
- A Trusted Storage Space contains ***Persistent Objects***. Each persistent object is identified by an ***Object Identifier***, which is a variable-length binary buffer from 0 to 64 bytes. Object identifiers can contain any bytes, including bytes corresponding to non-printable characters.
- A persistent object can be a ***Cryptographic Key Object***, a ***Cryptographic Key-Pair Object***, or a ***Data Object***.
- Each persistent object has a type, which precisely defines the content of the object. For example, there are object types for AES keys, RSA key-pairs, data objects, etc.
- All persistent objects have an associated ***Data Stream***. Data objects have only a data stream. Cryptographic objects (that is, keys or key-pairs) have a data stream, ***Object Attributes***, and metadata.
  - The Data Stream is entirely managed in the TA memory space. It can be loaded into a TA-allocated buffer when the object is opened or stored from a TA-allocated buffer when the object is created. It can also be accessed as a stream, so it can be used to store large amount of data accessed by small chunks.
  - Object Attributes are used for small amount of data (typically a few tens or hundreds of bytes). They can be stored in a memory pool that is separated from the TA instance and some attributes may be hidden from the TA itself. Attributes are used to store the key material in a structured way. For example, an RSA key-pair has an attribute for the modulus, the public exponent, the private exponent, etc. When an object is created, all Object Attributes are specified.

    Note that an Implementation is allowed to store more information in an object than the visible attributes. For example, some data might be pre-computed and stored internally to accelerate subsequent cryptographic operations.

  - The metadata associated with each cryptographic object includes:
    - ***Key Size*** in bits. The precise meaning depends on the key algorithm. For example, AES key can have 128 bits, 192 bits, or 256 bits; RSA keys can have 1024 bits or 2048 bits or any size below 2048 bits, etc.
    - ***Key usage flags***, which define the operations permitted with the key as well as whether the sensitive parts of the key material can be retrieved by the TA or not.
- A TA can also allocate ***Transient Objects***. Compared to persistent objects:
  - Transient objects are held in memory and are automatically wiped and reclaimed when they are closed or when the TA instance is destroyed.
  - Transient objects contain only attributes and no data stream.
  - A transient object can be ***uninitialized***, in which case it is an object container allocated with a certain object type and maximum size but with no attributes. A transient object becomes ***initialized*** when its attributes are populated. Note that persistent objects are always created initialized. This means that when the TA wants to generate or derive a persistent key, it has to first use a transient object then write the attributes of a transient object into a persistent object.
  - Transient objects have no identifier, they are only manipulated through object handles.
  - Currently, transient objects are used for cryptographic keys and key-pairs.
- Persistent and transient objects are manipulated through opaque ***Object Handles***.
  - Some functions accept both types of object handles. For example, a cryptographic operation can be started with either a transient key or a persistent key.
  - Some functions accept only handles on transient objects. For example, populating the attributes of an object works only with a transient object because it requires an uninitialized object and persistent objects are always fully initialized.
  - Finally, the file-like API functions to access the data stream work only with persistent objects because transient objects have no data stream.

Cryptographic operations are described in Chapter 6.

5.2 Data Types

5.2.1 TEE_Attribute

An array of this type is passed whenever a set of attributes must be specified to a function of the API.

```
typedef struct {
   uint32_t attributeID;
   union
   {
      struct
      {
         [inbuf] void* buffer; size_t length;
      } ref;
      struct
      {
         uint32_t a, b;
      } value;
   } content;
} TEE_Attribute;
```

An attribute can be either a buffer attribute or a value attribute. This is determined by bit [29] of the attribute identifier. If this bit is set to 0, then the attribute is a buffer attribute and the field `ref` must be selected. If the bit is set to 1, then it is a value attribute and the field `value` must be selected.

When an array of attributes is passed to a function, either to populate an object or to specify operation parameters, and if an attribute identifier is present twice in the array, then only the first occurrence is used.

5.2.2 TEE_ObjectInfo

```
typedef struct {
   uint32_t objectType;
   uint32_t objectSize;
   uint32_t maxObjectSize;
   uint32_t objectUsage;
   uint32_t dataSize;
   uint32_t dataPosition;
   uint32_t handleFlags;
} TEE_ObjectInfo;
```

See the documentation of function `TEE_GetObjectInfo` in section 5.4.1 for a description of this structure.

5.2.3 TEE_Whence

```
typedef enum
{
   TEE_DATA_SEEK_SET = 0,
   TEE_DATA_SEEK_CUR,
   TEE_DATA_SEEK_END
} TEE_Whence;
```

This structure enumerates the possible start offset when moving a data position in the data stream associated with a persistent object.

5.2.4 TEE_ObjectHandle

```
typedef struct __TEE_ObjectHandle* TEE_ObjectHandle
```

`TEE_ObjectHandle` is an opaque handle on a cryptographic object. These handles are returned by the function `TEE_GetObjectInfo` specified in section 5.4.1.

5.2.5 TEE_ObjectEnumHandle

```
typedef struct __TEE_ObjectEnumHandle* TEE_ObjectEnumHandle
```

`TEE_ObjectEnumHandle` is an opaque handle on an object enumerator. These handles are returned by the function `TEE_AllocatePersistentObjectEnumerator` specified in section 5.7.1.

5.3 Constants

Table 5-1: Object Storage Constants

Constant Name	Value
TEE_OBJECT_STORAGE_PRIVATE	0x00000001

Table 5-2: Data Flag Constants

Constant Name	Value
TEE_DATA_FLAG_ACCESS_READ	0x00000001
TEE_DATA_FLAG_ACCESS_WRITE	0x00000002
TEE_DATA_FLAG_ACCESS_WRITE_META	0x00000004
TEE_DATA_FLAG_SHARE_READ	0x00000010
TEE_DATA_FLAG_SHARE_WRITE	0x00000020
TEE_DATA_FLAG_CREATE	0x00000200
TEE_DATA_FLAG_EXCLUSIVE	0x00000400

Table 5-3: Usage Constants

Constant Name	Value
TEE_USAGE_EXTRACTABLE	0x00000001
TEE_USAGE_ENCRYPT	0x00000002
TEE_USAGE_DECRYPT	0x00000004
TEE_USAGE_MAC	0x00000008
TEE_USAGE_SIGN	0x00000010
TEE_USAGE_VERIFY	0x00000020
TEE_USAGE_DERIVE	0x00000040

Table 5-4: Handle Flag Constants

Constant Name	Value
TEE_HANDLE_FLAG_PERSISTENT	0x00010000
TEE_HANDLE_FLAG_INITIALIZED	0x00020000
TEE_HANDLE_FLAG_KEY_SET	0x00040000
TEE_HANDLE_FLAG_EXPECT_TWO_KEYS	0x00080000

Table 5-5: Operation Constants

Constant Name	Value
TEE_OPERATION_CIPHER	1
TEE_OPERATION_MAC	3
TEE_OPERATION_AE	4
TEE_OPERATION_DIGEST	5
TEE_OPERATION_ASYMMETRIC_CIPHER	6
TEE_OPERATION_ASYMMETRIC_SIGNATURE	7
TEE_OPERATION_KEY_DERIVATION	8

Table 5-6: Miscellaneous Constants

Constant Name	Value
TEE_DATA_MAX_POSITION	0xFFFFFFFF
TEE_OBJECT_ID_MAX_LEN	64

5.4 Generic Object Functions

These functions can be called on both transient and persistent object handles.

5.4.1 TEE_GetObjectInfo

```
void TEE_GetObjectInfo(
         TEE_ObjectHandle      object,
   [out] TEE_ObjectInfo* objectInfo
)
```

Description

The `TEE_GetObjectInfo` function returns the characteristics of an object. It fills in the following fields in the structure `TEE_ObjectInfo`:

- `objectType`: The parameter `objectType` passed when the object was created
- `objectSize`: Set to 0 for an uninitialized object
- `maxObjectSize`
  - For a persistent object, set to `objectSize`
  - For a transient object, set to the parameter `maxObjectSize` passed to `TEE_AllocateTransientObject`
- `objectUsage`: A bit vector of the `TEE_USAGE_XXX` bits defined in Table 5-3. Initially set to `0xFFFFFFFF`. Can be narrowed by calling `TEE_SetRestrictUsage`.
- `dataSize`
  - For a persistent object, set to the current size of the data associated with the object
  - For a transient object, always set to 0
- `dataPosition`
  - For a persistent object, set to the current position in the data for this handle. Data positions for different handles on the same object may differ.
  - For a transient object, set to 0
- `handleFlags`: A bit vector containing one or more of the following flags:
  - `TEE_HANDLE_FLAG_PERSISTENT`: Set for a persistent object
  - `TEE_HANDLE_FLAG_INITIALIZED`
    - For a persistent object, always set
    - For a transient object, initially cleared, then set when the object becomes initialized
  - `TEE_DATA_FLAG_XXX`: Only for persistent objects, the flags used to open or create the object

Parameters

- `object`: Handle of the object
- `objectInfo`: Pointer to a structure filled with the object information

Panic Reasons

- `object` is not a valid opened object handle.

5.4.2 TEE_RestrictObjectUsage

```
void TEE_RestrictObjectUsage(
   TEE_ObjectHandle  object,
   uint32_t          objectUsage
)
```

Description

The TEE_RestrictObjectUsage function restricts the object usage flags of an object handle to contain at most the flags passed in the objectUsage parameter.

For each bit in the parameter objectUsage:

- If the bit is set to 1, the corresponding usage flag in the object is left unchanged.
- If the bit is set to 0, the corresponding usage flag in the object is cleared.

For example, if the usage flags of the object are set to TEE_USAGE_ENCRYPT | TEE_USAGE_DECRYPT and if objectUsage is set to TEE_USAGE_ENCRYPT | TEE_USAGE_EXTRACTABLE, then the only remaining usage flag in the object after calling the function TEE_RestrictObjectUsage is TEE_USAGE_ENCRYPT.

Note that an object usage flag can only be cleared. Once it is cleared, it cannot be set to 1 again until the whole object is reset using the function TEE_ResetObject. For a transient object, resetting the object also clears all the key material stored in the container.

For a persistent object, setting the object usage MUST be an atomic operation.

Parameters

- object: Handle on an object
- objectUsage: New object usage, an OR combination of one or more of the TEE_USAGE_XXX constants defined in Table 5-3

Panic Reasons

- object is not a valid opened object handle.

5.4.3 TEE_GetObjectBufferAttribute

```
TEE_Result TEE_GetObjectBufferAttribute(
               TEE_ObjectHandle   object,
               uint32_t     attributeID,
   [outbuf]    void*        buffer, size_t* size
)
```

Description

The `TEE_GetObjectBufferAttribute` function extracts one buffer attribute from an object.

The attribute is identified by the argument `attributeID`. The precise meaning of this parameter depends on the container type and size and is defined in section 6.11.

Bit [29] of the attribute identifier must be set to 0, i.e., it must denote a buffer attribute.

They are two kinds of object attributes:

- Public object attributes can always be extracted whatever the status of the container.
- Secret attributes can be extracted only if the object's key usage contains the `TEE_USAGE_EXTRACTABLE` flag.

See section 6.11 for a definition of all available object attributes and their level of protection.

Parameters

- `object`: Handle of the object
- `attributeID`: Identifier of the attribute to retrieve
- `buffer`, `size`: Output buffer to get the content of the attribute

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_ITEM_NOT_FOUND`: If the attribute is not found on this object or if the object is a transient uninitialized object
- `TEE_ERROR_SHORT_BUFFER`: If `buffer` is `NULL` or too small to contain the key part

Panic Reasons

- `object` is not a valid opened object handle.
- The object is not initialized.
- Bit [29] of `attributeID` is not set to 0, so the attribute is not a buffer attribute.
- Bit [28] of `attributeID` is set to 1, denoting a protected attribute, and the object usage does not contain the `TEE_USAGE_EXTRACTABLE` flag.

5.4.4 TEE_GetObjectValueAttribute

```
TEE_Result TEE_GetObjectValueAttribute(
               TEE_ObjectHandle   object,
               uint32_t     attributeID,
   [outopt]    uint32_t*    a,
   [outopt]    uint32_t*    b
)
```

Description

The `TEE_GetObjectValueAttribute` function extracts a value attribute from an object.

The attribute is identified by the argument `attributeID`. The precise meaning of this parameter depends on the container type and size and is defined in section 6.11.

Bit [29] of the attribute identifier must be set to 1, i.e., it must denote a value attribute.

They are two kinds of object attributes:

- Public object attributes can always be extracted whatever the status of the container.
- Secret attributes can be extracted only if the object's key usage contains the `TEE_USAGE_EXTRACTABLE` flag.

See section 6.11 for a definition of all available object attributes and their level of protection.

Parameters

- `object`: Handle of the object
- `attributeID`: Identifier of the attribute to retrieve
- `a`, `b`: Pointers on the placeholders filled with the attribute field `a` and `b`. Each can be `NULL` if the corresponding field is not of interest to the caller.

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_ITEM_NOT_FOUND`: If the attribute is not found on this object or if the object is a transient uninitialized object
- `TEE_ERROR_ACCESS_DENIED`: For an attempt to extract a secret part of a non-extractable container

Panic Reasons

- `object` is not a valid opened object handle.
- The object is not initialized.
- Bit [29] of `attributeID` is not set to 1, so the attribute is not a value attribute.
- Bit [28] of `attributeID` is set to 1, denoting a protected attribute, and the object usage does not contain the `TEE_USAGE_EXTRACTABLE` flag.

5.4.5 TEE_CloseObject

```
void TEE_CloseObject( TEE_ObjectHandle object)
```

Description

The `TEE_CloseObject` function closes an opened object handle. The object can be persistent or transient.

Parameters

- `object`: Handle on the object to close. If set to `TEE_HANDLE_NULL`, does nothing.

Panic Reasons

- `object` is not a valid opened object handle and is not equal to `TEE_HANDLE_NULL`.

5.5 Transient Object Functions

5.5.1 TEE_AllocateTransientObject

```
TEE_Result TEE_AllocateTransientObject(
         uint32_t     objectType,
         uint32_t           maxObjectSize,
   [out] TEE_ObjectHandle*  object
)
```

Description

The `TEE_AllocateTransientObject` function allocates an uninitialized transient object, i.e., a container for attributes. Transient objects are used to hold a cryptographic object (key or key-pair). The object type and the maximum object characteristic size must be specified so that all the container resources can be pre-allocated.

As allocated, the container is uninitialized. It can be initialized by subsequently importing the object material, generating an object, deriving an object, or loading an object from the Trusted Storage.

The initial value of the key usage associated with the container is `0xFFFFFFFF`, which means that it contains all usage flags. You can use the function `TEE_RestrictObjectUsage` to restrict the usage of the container.

The returned handle is used to refer to the newly-created container in all subsequent functions that require an object container: key management and operation functions. The handle remains valid until the container is deallocated using the function `TEE_FreeTransientObject`.

As shown in Table 5-7, the object type determines the possible object size.

Table 5-7: TEE_AllocateTransientObject and Object Sizes

Object Type	Possible Object Sizes
TEE_TYPE_AES	128, 192, or 256 bits
TEE_TYPE_DES	Always 56 bits
TEE_TYPE_DES3	112 or 168 bits
TEE_TYPE_HMAC_MD5	Between 64 and 512 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA1	Between 80 and 512 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA224	Between 112 and 512 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA256	Between 192 and 1024 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA384	Between 256 and 1024 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA512	Between 256 and 1024 bits, multiple of 8 bits
TEE_TYPE_RSA_PUBLIC_KEY	Object size is the number of bits in the modulus. All key size up to 2048 bits must be supported. Support for bigger key sizes is implementation-dependent. Minimum key size is 256 bits.[3]
TEE_TYPE_RSA_KEYPAIR	Same as for RSA public key size.
TEE_TYPE_DSA_PUBLIC_KEY	Between 512 and 1024 bits, multiple of 64 bits
TEE_TYPE_DSA_KEYPAIR	
TEE_TYPE_DH_KEYPAIR	From 256 to 2048 bits
TEE_TYPE_GENERIC_SECRET	Multiple of 8 bits, up to 4096 bits. This type is intended for secret data that is not directly used as a key in a cryptographic operation, but participates in a key derivation.

Parameters

- `objectType`: Type of uninitialized object container to be created
- `maxObjectSize`: Size of the object. The interpretation of this parameter depends on the object type and is defined in Table 5-7 above.
- `object`: Filled with a handle on the newly created key container

Return Value

- `TEE_SUCCESS`: On success
- `TEE_ERROR_OUT_OF_MEMORY`: If not enough resources are available to allocate the object handle
- `TEE_ERROR_NOT_SUPPORTED`: If the object size is not supported. Note that a compliant Implementation must implement all the object types, algorithms, and object sizes described in Table 5-7 above; support for other sizes or algorithms is implementation-defined.

[3] Note that using RSA keys smaller than 1024 bits is nowadays considered insecure but may be required for legacy protocols.

5.5.2 TEE_FreeTransientObject

```
void TEE_FreeTransientObject(
   TEE_ObjectHandle object
)
```

Description

The `TEE_FreeTransientObject` function deallocates a transient object previously allocated with `TEE_AllocateTransientObject`. After this function has been called, the object handle is no longer valid and all resources associated with the transient object must have been reclaimed.

If the object is initialized, the object attributes are cleared before the object is deallocated.

This function cannot fail. It does nothing if `object` is `TEE_HANDLE_NULL`.

Parameters

- `object`: Handle on the object to free

Panic Reasons

- `object` is not a valid opened object handle and is not equal to `TEE_HANDLE_NULL`.

5.5.3 TEE_ResetTransientObject

```
void TEE_ResetTransientObject(
   TEE_ObjectHandle   object
)
```

Description

The `TEE_ResetTransientObject` function resets a transient object to its initial state after allocation.

If the object is currently initialized, the function clears the object of all its material. The object is then uninitialized again.

In any case, the function resets the key usage of the container to `0xFFFFFFFF`.

This function does nothing if `object` is set to `TEE_HANDLE_NULL`.

Parameters

- `object`: Handle on a transient object to reset

Panic Reasons

- `object` is not a valid opened object handle and is not equal to `TEE_HANDLE_NULL`.

5.5.4 TEE_PopulateTransientObject

```
TEE_Result TEE_PopulateTransientObject(
        TEE_ObjectHandle      object,
   [in] TEE_Attribute*        attrs, uint32_t attrCount
)
```

Description

The `TEE_PopulateTransientObject` function populates an uninitialized object container with object attributes passed by the TA in the `attrs` parameter.

When this function is called, the object must be uninitialized. If the object is initialized, the caller must first clear it using the function `TEE_ClearObject`.

Note that if the object type is a key-pair, then this function sets both the private and public parts of the key-pair.

As shown in Table 5-8, the interpretation of the `attrs` parameter depends on the object type.

Table 5-8: TEE_PopulateTransientObject: Supported Attributes

Object Type	Parts
TEE_TYPE_AES	For all secret key objects, the TEE_ATTR_SECRET_VALUE must be provided. Other parts are ignored.
TEE_TYPE_DES	
TEE_TYPE_DES3	
TEE_TYPE_HMAC_MD5	
TEE_TYPE_HMAC_SHA1	
TEE_TYPE_HMAC_SHA224	
TEE_TYPE_HMAC_SHA256	
TEE_TYPE_HMAC_SHA384	
TEE_TYPE_HMAC_SHA512	
TEE_TYPE_GENERIC_SECRET	
TEE_TYPE_RSA_PUBLIC_KEY	The following parts must be provided: TEE_ATTR_RSA_MODULUS TEE_ATTR_RSA_PUBLIC_EXPONENT
TEE_TYPE_RSA_KEYPAIR	The following parts must be provided: TEE_ATTR_RSA_MODULUS TEE_ATTR_RSA_PUBLIC_EXPONENT TEE_ATTR_RSA_PRIVATE_EXPONENT The CRT parameters are optional. If any of these parts is provided, then all of them must be provided: TEE_ATTR_RSA_PRIME1 TEE_ATTR_RSA_PRIME2 TEE_ATTR_RSA_EXPONENT1 TEE_ATTR_RSA_EXPONENT2 TEE_ATTR_RSA_COEFFICIENT

Object Type	Parts
TEE_TYPE_DSA_PUBLIC_KEY	The following parts must be provided: TEE_ATTR_DSA_PRIME TEE_ATTR_DSA_SUBPRIME TEE_ATTR_DSA_BASE TEE_ATTR_DSA_PUBLIC_VALUE
TEE_TYPE_DSA_KEYPAIR	The following parts must be provided: TEE_ATTR_DSA_PRIME TEE_ATTR_DSA_SUBPRIME TEE_ATTR_DSA_BASE TEE_ATTR_DSA_PRIVATE_VALUE TEE_ATTR_DSA_PUBLIC_VALUE
TEE_TYPE_DH_KEYPAIR	The following parts must be provided: TEE_ATTR_DH_PRIME TEE_ATTR_DH_BASE TEE_ATTR_DH_PUBLIC_VALUE TEE_ATTR_DH_PRIVATE_VALUE Optionally, TEE_ATTR_DH_SUBPRIME may be provided, too.

All mandatory attributes must be specified, otherwise it is a Panic Reason.

The Implementation can detect whether the parameters are consistent; for example, if the numbers supposed to be prime are indeed prime. However, it is not required to do these checks fully and reliably. If it detects invalid attributes, it MUST return the error code TEE_ERROR_BAD_PARAMETERS and MUST NOT panic. If it does not detect any inconsistencies, it MUST be able to later proceed with all operations associated with the object without error. In this case, it is not required to make sensible computations, but all computations must terminate and output some result.

Parameters

- `object`: Handle on an already created transient and uninitialized object
- `attrs`, `attrCount`: Array of object attributes

Return Value

- TEE_SUCCESS: In case of success. In this case, the content of the object MUST be initialized.
- TEE_ERROR_BAD_PARAMETERS: If an incorrect or inconsistent attribute value is detected. In this case, the content of the object container MUST remain uninitialized.

Panic Reasons

- `object` is not a valid opened object handle that is transient and uninitialized.
- Some mandatory attribute is missing.

5.5.5 TEE_InitRefAttribute, TEE_InitValueAttribute

```
void TEE_InitRefAttribute(
   [out]   TEE_Attribute* attr,
           uint32_t       attributeID
   [inbuf] void*          buffer, size_t length
)
```

```
void  TEE_InitValueAttribute(
   [out]   TEE_Attribute* attr,
           uint32_t       attributeID
           uint32_t       a, b
)
```

Description

The `TEE_InitRefAttribute` and `TEE_InitValueAttribute` helper functions can be used to populate a single attribute either with a reference to a buffer or with integer values.

For example, the following code can be used to initialize a DH key generation:

```
TEE_Attribute attrs[3];
TEE_InitRefAttribute(&attrs[0], TEE_ATTR_DH_PRIME, &p, len);
TEE_InitRefAttribute(&attrs[1], TEE_ATTR_DH_BASE, &g, len);
TEE_InitValueAttribute(&attrs[2], TEE_ATTR_DH_X_BITS, xBits, 0);
TEE_GenerateKey(key, 1024, attrs, sizeof(attrs)/sizeof(TEE_Attribute));
```

Note that in the case of `TEE_InitRefAttribute`, only the buffer pointer is copied, not the content of the buffer.

Panic Reasons

- Bit [29] of `attributeID` describing whether the attribute identifier is a value or reference is not consistent with the function.

5.5.6 TEE_CopyObjectAttributes

```
void TEE_CopyObjectAttributes(
   TEE_ObjectHandle destObject,
   TEE_ObjectHandle srcObject
)
```

Description

The TEE_CopyObjectAttributes function populates an uninitialized object handle with the attributes of another object handle; that is, it populates the attributes of destObject with the attributes of srcObject. It is most useful in the following situations:

- To extract the public key attributes from a key-pair object
- To copy the attributes from a persistent object into a transient object

destObject must refer to an uninitialized object handle and must therefore be a transient object.

The source and destination objects must have compatible types and sizes in the following sense:

- The type of destObject must be a subtype of srcObject, i.e., one of the following must be true:
  - The type of destObject is equal to the type of srcObject.
  - The type of destObject is TEE_TYPE_RSA_PUBLIC_KEY and the type of srcObject is TEE_TYPE_RSA_KEYPAIR.
  - The type of destObject is TEE_TYPE_DSA_PUBLIC_KEY and the type of srcObject is TEE_TYPE_DSA_KEYPAIR.
- The size of srcObject must be less than or equal to the maximum size of destObject.

The effect of this function on destObject is identical to the function TEE_PopulateTransientObject except that the attributes are taken from srcObject instead of from parameters.

The object usage of destObject is set to the bitwise AND of the current object usage of destObject and the object usage of srcObject.

This function cannot fail but may panic if the source and destination objects are not compatible or are not in the correct state.

Parameters

- destObject: Handle on an uninitialized transient object
- srcObject: Handle on an initialized object

Panic Reasons

- srcObject is not initialized.
- destObject is not uninitialized.
- The type and size of srcObject and destObject are not compatible.

5.5.7 TEE_GenerateKey

```
TEE_Result TEE_GenerateKey(
        TEE_ObjectHandle  object,
        uint32_t          keySize,
   [in] TEE_Attribute*    params, uint32_t paramCount,
)
```

Description

The TEE_GenerateKey function generates a random key or a key-pair and populates a transient key object with the generated key material.

The size of the desired key is passed in the keySize parameter and must be less than the maximum size of the transient object.

As shown in Table 5-9, the generation algorithm can take parameters depending on the object type.

Table 5-9: TEE_GenerateKey Parameters

Object Type	Details
TEE_TYPE_AES	No parameter is necessary. The function generates the attribute TEE_ATTR_SECRET_VALUE.
TEE_TYPE_DES	
TEE_TYPE_DES3	
TEE_TYPE_HMAC_MD5	
TEE_TYPE_HMAC_SHA1	
TEE_TYPE_HMAC_SHA224	
TEE_TYPE_HMAC_SHA256	
TEE_TYPE_HMAC_SHA384	
TEE_TYPE_HMAC_SHA512	
TEE_TYPE_GENERIC_SECRET	
TEE_TYPE_RSA_KEYPAIR	No parameter is required. The function generates and populates the following attributes: • TEE_ATTR_RSA_MODULUS • TEE_ATTR_RSA_PUBLIC_EXPONENT • TEE_ATTR_RSA_PRIVATE_EXPONENT • TEE_ATTR_RSA_PRIME1 • TEE_ATTR_RSA_PRIME2 • TEE_ATTR_RSA_EXPONENT1 • TEE_ATTR_RSA_EXPONENT2 • TEE_ATTR_RSA_COEFFICIENT

Object Type	Details
TEE_TYPE_DSA_KEYPAIR	The following domain parameter must be passed to the function • TEE_ATTR_DSA_PRIME • TEE_ATTR_DSA_SUBPRIME • TEE_ATTR_DSA_BASE The function generates and populates the following attributes: • TEE_ATTR_DSA_PUBLIC_VALUE • TEE_ATTR_DSA_PRIVATE_VALUE
TEE_TYPE_DH_KEYPAIR	The following domain parameters must be passed to the function • TEE_ATTR_DH_PRIME • TEE_ATTR_DH_BASE The following parameters can optionally be passed: • TEE_ATTR_DH_SUBPRIME (*q*): If present, constrains the private value *x* to be in the range [2, *q*-2] • TEE_ATTR_DH_X_BITS (ℓ): If present, constrains the private value *x* to have ℓ bits • If neither of these optional parts is specified, then the only constraint on *x* is that it is less than *p*-1. The function generates and populates the following attributes: • TEE_ATTR_DH_PUBLIC_VALUE • TEE_ATTR_DH_PRIVATE_VALUE • TEE_ATTR_DH_X_BITS (number of bits in *x*)

Once the key material has been generated, the transient object is populated exactly as in the function TEE_PopulateTransientObject except that the key material is randomly generated internally instead of being passed by the caller.

Parameters

- object: Handle on an uninitialized transient key to populate with the generated key
- keySize: Requested key size. Must be less than or equal to the maximum size of the object container.
- params, paramCount: Parameters for the key generation

Return Value

- TEE_SUCCESS: On success
- TEE_ERROR_BAD_PARAMETERS: If an incorrect or inconsistent attribute is detected

Panic Reasons

- object is not a valid opened object handle that is transient and uninitialized.
- keySize is too large.
- A mandatory parameter is missing.

5.6 Persistent Object Functions

5.6.1 TEE_OpenPersistentObject

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

Description

The `TEE_OpenPersistentObject` function opens a handle on an existing persistent object. It returns a handle that can be used to access the object's attributes and data stream.

The `storageID` parameter indicates which Trusted Storage Space to access. Possible values are:

- `TEE_STORAGE_PRIVATE`: This refers to the private Trusted Storage of the current Trusted Application. This storage space is accessible to all instances of this Trusted Application.

The `flags` parameter is a set of flags that controls the access rights and sharing permissions with which the object handle is opened. The value of the `flags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- Access control flags:
  - `TEE_DATA_FLAG_ACCESS_READ`: The object is opened with the read access right. This allows the Trusted Application to call the function `TEE_ReadObjectData`.
  - `TEE_DATA_FLAG_ACCESS_WRITE`: The object is opened with the write access right. This allows the Trusted Application to call the functions `TEE_WriteObjectData` and `TEE_TruncateObjectData`.
  - `TEE_DATA_FLAG_ACCESS_WRITE_META`: The object is opened with the write-meta access right. This allows the Trusted Application to call the functions `TEE_CloseAndDeletePersistentObject` and `TEE_RenamePersistentObject`.
- Sharing permission control flags:
  - `TEE_DATA_FLAG_SHARE_READ`: The caller allows another handle on the object to be created with read access.
  - `TEE_DATA_FLAG_SHARE_WRITE`: The caller allows another handle on the object to be created with write access.
- Other flags are reserved for future use and should be set to 0.

Multiple handles may be opened on the same object simultaneously, but sharing must be explicitly allowed as described in section 5.6.3.

The initial data position in the data stream is set to 0.

Parameters

- `storageID`: The storage to use. It must be `TEE_STORAGE_PRIVATE`.
- `objectID`, `objectIDLen`: The object identifier. Note that this buffer cannot reside in shared memory.
- `flags`: The flags which determine the settings under which the object is opened.
- `object`: A pointer to the handle, which contains the opened handle upon successful completion. If this function fails for any reason, the value pointed to by `object` is set to `TEE_HANDLE_NULL`. When the object handle is no longer required, it must be closed using a call to the `TEE_CloseObject` function.

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_ITEM_NOT_FOUND`: If the storage denoted by `storageID` does not exist or if the object identifier cannot be found in the storage
- `TEE_ERROR_ACCESS_CONFLICT`: If an access right conflict was detected while opening the object
- `TEE_ERROR_OUT_OF_MEMORY`: If there is not enough memory to complete the operation

Panic Reasons

- `objectIDLen` is greater than `TEE_OBJECT_ID_MAX_LEN`.

5.6.2 TEE_CreatePersistentObject

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

Description

The `TEE_CreatePersistentObject` function creates a persistent object with initial attributes and an initial data stream content, and optionally returns a handle on the created object..

The `storageID` parameter indicates which Trusted Storage Space to access. Possible values are:

- `TEE_STORAGE_PRIVATE`: This refers to the private Trusted Storage of the current Trusted Application. This storage space is accessible to all instances of this Trusted Application.

The `flags` parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the `flags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- Access control flags:
  - `TEE_DATA_FLAG_ACCESS_READ`: The object is opened with the read access right. This allows the Trusted Application to call the function `TEE_ReadObjectData`.
  - `TEE_DATA_FLAG_ACCESS_WRITE`: The object is opened with the write access right. This allows the Trusted Application to call the functions `TEE_WriteObjectData` and `TEE_TruncateObjectData`.
  - `TEE_DATA_FLAG_ACCESS_WRITE_META`: The object is opened with the write-meta access right. This allows the Trusted Application to call the functions `TEE_CloseAndDeletePersistentObject` and `TEE_RenamePersistentObject`.
- Sharing permission control flags:
  - `TEE_DATA_FLAG_SHARE_READ`: The caller allows another handle on the object to be created with read access.
  - `TEE_DATA_FLAG_SHARE_WRITE`: The caller allows another handle on the object to be created with write access.
- `TEE_DATA_FLAG_EXCLUSIVE`: If the object does not already exist, it is created. Otherwise, the error `TEE_ERROR_ACCESS_CONFLICT` is returned.
- Other flags are reserved for future use and should be set to 0.

The attributes of the newly created persistent object are taken from `attributes`, which can be another persistent object or an uninitialized transient object. The object type, size, and usage are also copied from `attributes`.

Multiple handles may be opened on the same object simultaneously, but sharing must be explicitly allowed as described in section 5.6.3.

The initial data position in the data stream is set to 0.

Parameters

- `storageID`: The storage to use. It must be `TEE_STORAGE_PRIVATE`.
- `objectID`, `objectIDLen`: The object identifier. Note that this cannot reside in shared memory.
- `flags`: The flags which determine the settings under which the object is opened
- `attributes`: A handle on a transient object from which to take the persistent object attributes. Can be `TEE_HANDLE_NULL` if the persistent object contains no attribute, for example if it is a pure data object.
- `initialData`, `initialDataLen`: The initial data content of the persistent object
- `object`: A pointer to the handle, which contains the opened handle upon successful completion. If this function fails for any reason, the value pointed to by `object` is set to `TEE_HANDLE_NULL`. When the object handle is no longer required, it must be closed using a call to the `TEE_CloseObject` function.

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_ITEM_NOT_FOUND`: If the storage denoted by `storageID` does not exist or if the object identifier cannot be found in the storage
- `TEE_ERROR_ACCESS_CONFLICT`: If an access right conflict was detected while opening the object
- `TEE_ERROR_OUT_OF_MEMORY`: If there is not enough memory to complete the operation
- `TEE_ERROR_STORAGE_NO_SPACE`: If insufficient space is available to create the persistent object

Panic Reasons

- `objectIDLen` is greater than `TEE_OBJECT_ID_MAX_LEN`.
- `attributes` is not `TEE_HANDLE_NULL` and is not a valid handle on an initialized object containing the type and attributes of the persistent object to create.

5.6.3 Persistent Object Sharing Rules

Multiple handles may be opened on the same object simultaneously using the functions TEE_OpenPersistentObject or TEE_CreatePersistentObject, but sharing must be explicitly allowed. More precisely, at any one time the following constraints apply: If more than one handle is opened on the same object, and if any of these object handles was opened with the flag TEE_DATA_FLAG_ACCESS_READ, then all the object handles must have been opened with the flag TEE_DATA_FLAG_SHARE_READ. There is a corresponding constraint with the flags TEE_DATA_FLAG ACCESS_WRITE and TEE_DATA_FLAG_SHARE_WRITE. Accessing an object with write-meta rights is exclusive and can never be shared.

When one of the functions TEE_OpenPersistentObject or TEE_CreatePersistentObject is called and if opening the object would violate these constraints, then the function returns the error code TEE_ERROR_ACCESS_CONFLICT.

Any bits in `flags` not defined in section 5.3 are reserved for future use and must be set to zero.

The examples in Table 5-10 illustrate the behavior of the TEE_OpenPersistentObject function when called twice on the same object. Note that for readability, the flag names used in Table 5-10 have been abbreviated by removing the 'TEE_DATA_FLAG_' prefix from their name, and any non-TEE_SUCCESS error codes have been shortened by removing the 'TEE_ERROR_' prefix.

Table 5-10: TEE_OpenPersistentObject Sharing Rules

Value of `flags` for first open/create	Value of `flags` for second open/create	Return Code of second open/create	Comments
ACCESS_READ	ACCESS_READ	ACCESS_CONFLICT	The object handles have not been opened with the flag SHARE_READ. Only the first call will succeed.
ACCESS_READ | SHARE_READ	ACCESS_READ	ACCESS_CONFLICT	Not all the object handles have been opened with the flag SHARE_READ. Only the first call will succeed.
ACCESS_READ | SHARE_READ	ACCESS_READ | SHARE_READ	TEE_SUCCESS	All the object handles have been opened with the flag SHARE_READ.
ACCESS_READ	ACCESS_WRITE	ACCESS_CONFLICT	Objects are not opened with share flags. Only the first call will succeed.
ACCESS_READ | SHARE_READ | SHARE_WRITE	ACCESS_WRITE | SHARE_READ | SHARE_WRITE	TEE_SUCCESS	All the object handles have been opened with the share flags.
ACCESS_READ | SHARE_READ | ACCESS_WRITE | SHARE_WRITE	ACCESS_WRITE_META	ACCESS_CONFLICT	The write-meta flag indicates an exclusive access to the object. Only the first call will succeed.

Value of `flags` for first open/create	Value of `flags` for second open/create	Return Code of second open/create	Comments
`SHARE_READ`	`ACCESS_WRITE | SHARE_WRITE`	`ACCESS_CONFLICT`	An object can be opened with only share flags, which locks the access to an object against a given mode. Here the first call prevents subsequent accesses in write mode.
`0`	`ACCESS_READ | SHARE_READ`	`ACCESS_CONFLICT`	An object can be opened with no flag set, which completely locks all subsequent attempts to access the object. Only the first call will succeed.

5.6.4 TEE_CloseAndDeletePersistentObject

```
void TEE_CloseAndDeletePersistentObject( TEE_ObjectHandle object )
```

Description

The `TEE_CloseAndDeletePersistentObject` function marks an object for deletion and closes the object handle.

The object handle must have been opened with the write-meta access right, which means access to the object is exclusive.

Deleting an object is atomic; once this function returns, the object is definitely deleted and no more open handles for that object exist.

If `object` is `TEE_HANDLE_NULL`, the function does nothing.

Parameters

- `object`: The object handle

Panic Reasons

- `object` is not a valid handle on a persistent object opened with the write-meta access right.

5.6.5 TEE_RenamePersistentObject

```
TEE_Result TEE_RenamePersistentObject(
           TEE_ObjectHandle  object,
     [in(newObjectIDLen)]    void*    newObjectID, size_t newObjectIDLen
 )
```

Description

The function TEE_RenamePersistentObject changes the identifier of an object. The object handle must have been opened with the write-meta access right, which means access to the object is exclusive.

Renaming an object is an atomic operation; either the object is renamed or nothing happens.

Parameters

- object: The object handle
- newObjectID, newObjectIDLen: A buffer containing the new object identifier. The identifier can contain arbitrary bytes, including the zero byte. The identifier length can be zero but must be less than or equal to TEE_OBJECT_ID_MAX_LEN. The buffer containing the new object identifier cannot reside in shared memory.

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ACCESS_CONFLICT: If an object with the same identifier already exists

Panic Reasons

- object is not a valid handle on a persistent object that has been opened with the write-meta access right.
- newObjectIDLen is more than TEE_OBJECT_ID_MAX_LEN.

5.7 Persistent Object Enumeration Functions

5.7.1 TEE_AllocatePersistentObjectEnumerator

```
TEE_Result TEE_AllocatePersistentObjectEnumerator(
      [out] TEE_ObjectEnumHandle* objectEnumerator )
```

Description

The TEE_AllocatePersistentObjectEnumerator function allocates a handle on an object enumerator. Once an object enumerator handle has been allocated, it can be reused for multiple enumerations.

Parameters

- objectEnumerator: A pointer filled with the newly-allocated object enumerator handle on success. Set to TEE_HANDLE_NULL in case of error.

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_OUT_OF_MEMORY: If there is not enough memory to allocate the enumerator handle

5.7.2 TEE_FreePersistentObjectEnumerator

```
void TEE_FreePersistentObjectEnumerator(TEE_ObjectEnumHandle
objectEnumerator )
```

Description

The TEE_FreePersistentObjectEnumerator function deallocates all resources associated with an object enumerator handle. After this function is called, the handle is no longer valid.

This function cannot fail.

Parameters

- objectEnumerator: The handle to close. If objectEnumerator is TEE_HANDLE_NULL, then this function does nothing.

Panic Reasons

- objectEnumerator is not a valid handle on an object enumerator.

5.7.3 TEE_ResetPersistentObjectEnumerator

```
void TEE_ResetPersistentObjectEnumerator(TEE_ObjectEnumHandle
objectEnumerator )
```

Description

The `TEE_ResetPersistentObjectEnumerator` function resets an object enumerator handle to its initial state after allocation. If an enumeration has been started, it is stopped.

This function cannot fail. It does nothing if `objectEnumerator` is `TEE_HANDLE_NULL`.

Parameters

- `objectEnumerator`: The handle to reset

Panic Reasons

- `objectEnumerator` is not `TEE_HANDLE_NULL` and is not a valid handle on an object enumerator.

5.7.4 TEE_StartPersistentObjectEnumerator

```
TEE_Result TEE_StartPersistentObjectEnumerator(
           TEE_ObjectEnumHandle objectEnumerator,
           uint32_t   storageID
)
```

Description

The TEE_StartPersistentObjectEnumerator function starts the enumeration of all the persistent objects in a given Trusted Storage. The object information can be retrieved by calling the function TEE_GetNextPersistentObject repeatedly.

The enumeration does not necessarily reflect a given consistent state of the storage: During the enumeration, other TAs or other instances of the TA may create, delete, or rename objects.

To stop an enumeration, the TA can call the function TEE_ResetPersistentObjectEnumerator, which detaches the enumerator from the Trusted Storage. The TA can call the function TEE_FreePersistentObjectEnumerator to completely deallocate the object enumerator.

If this function is called while an enumeration has already started, the enumeration is first reset then started.

Parameters

- objectEnumerator: A valid handle on an object enumerator
- storageID: The identifier of the storage in which the objects must be enumerated. Possible values are:
  - TEE_STORAGE_PRIVATE: Indicates the storage is private to the Trusted Application

Return Value

- TEE_SUCCESS: In case of success
- TEE_ITEM_NOT_FOUND: If the storage identifier is not TEE_STORAGE_PRIVATE or if there is no object in the TEE_STORAGE_PRIVATE store

Panic Reasons

- objectEnumerator is not a valid handle on an object enumerator.

5.7.5 TEE_GetNextPersistentObject

```
TEE_Result TEE_GetNextPersistentObject(
           TEE_ObjectEnumHandle      objectEnumerator,
    [out]  TEE_ObjectInfo  objectInfo,
    [out]  void*             objectID,
    [out]  size_t*           objectIDLen )
```

Description

The TEE_GetNextPersistentObject function gets the next object in an enumeration and returns information about the object: type, size, identifier, etc.

If there are no more objects in the enumeration or if there is no enumeration started, then the function returns TEE_ERROR_ITEM_NOT_FOUND.

Parameters

- objectEnumerator: A handle on the object enumeration
- objectInfo: A pointer to a TEE_ObjectInfo filled with the object information as specified in the function TEE_GetObjectInfo in section 5.4.1
- objectID: Pointer to an array of TEE_OBJECT_ID_MAX_LEN bytes filled with the object identifier
- objectIDLen: Filled with the size of the object identifier (from 0 to TEE_OBJECT_ID_MAX_LEN)

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_ITEM_NOT_FOUND: If there are no more elements in the object enumeration or if no enumeration is started on this handle

Panic Reasons

- objectEnumerator is not a valid handle on an object enumerator.

5.8 Data Stream Access Functions

These functions can be used to access the data stream of persistent objects. They work like a file API.

5.8.1 TEE_ReadObjectData

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

Description

The `TEE_ReadObjectData` function attempts to read `size` bytes from the data stream associated with the object `object` into the buffer pointed to by `buffer`.

The object handle must have been opened with the read access right.

The bytes are read starting at the position in the data stream currently stored in the object handle. The handle's position is incremented by the number of bytes actually read.

On completion `TEE_ReadObjectData` sets the number of bytes actually read in the `uint32_t` pointed to by `count`. The value written to `*count` may be less than `size` if the number of bytes until the end-of-stream is less than `size`. It is set to 0 if the position at the start of the read operation is at or beyond the end-of-stream. These are the only cases where `*count` may be less than `size`.

No data transfer can occur past the current end of stream. If an attempt is made to read past the end-of-stream, the `TEE_ReadObjectData` function stops reading data at the end-of-stream and returns the data read up to that point. This is still a success. The position indicator is then set at the end-of-stream. If the position is at, or past, the end of the data when this function is called, then no bytes are copied to `*buffer` and `*count` is set to 0.

Parameters

- `object`: The object handle
- `buffer`: A pointer to the memory which, upon successful completion, contains the bytes read
- `size`: The number of bytes to read
- `count`: A pointer to the variable which upon successful completion contains the number of bytes read

Return Value

The only possible return value is `TEE_SUCCESS`. The presence of an error return value is for future versions of the specification.

Panic Reasons

- `object` is not a valid handle on a persistent object opened with the read access right.

5.8.2 TEE_WriteObjectData

```
TEE_Result TEE_WriteObjectData(
            TEE_ObjectHandle   object,
       [in] void*        buffer, size_t size )
```

Description

The `TEE_WriteObjectData` function writes `size` bytes from the buffer pointed to by `buffer` to the data stream associated with the open object handle `object`.

The object handle must have been opened with the write access permission.

If the current data position points before the end-of-stream, then `size` bytes are written to the data stream, overwriting bytes starting at the current data position. If the current data position points beyond the stream's end, then the data stream is first extended with zero bytes until the length indicated by the data position indicator is reached, and then `size` bytes are written to the stream. Thus, the size of the data stream can be increased as a result of this operation.

The data position indicator is advanced by `size`. The data position indicators of other object handles opened on the same object are not changed.

Writing in a data stream is atomic; either the entire operation completes successfully or no write is done.

Parameters

- `object`: The object handle
- `buffer`: The buffer containing the data to be written
- `size`: The number of bytes to write

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_STORAGE_NO_SPACE`: If insufficient storage space is available

Panic Reasons

- `object` is not a valid handle on a persistent object opened with the write access right.

5.8.3 TEE_TruncateObjectData

```
TEE_Result TEE_TruncateObjectData(
           TEE_ObjectHandle object,
           uint32_t size )
```

Description

The function TEE_TruncateObjectData changes the size of a data stream. If size is less than the current size of the data stream then all bytes beyond size are removed. If size is greater than the current size of the data stream then the data stream is extended by adding zero bytes at the end of the stream.

The object handle must have been opened with the write access permission.

This operation does not change the data position of any handle opened on the object. Note that if the current data position of such a handle is beyond size, the data position will point beyond the object data's end after truncation.

Truncating a data stream is atomic: Either the data stream is successfully truncated or nothing happens.

Parameters

- object: The object handle
- size: The new size of the data stream

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_STORAGE_NO_SPACE: If insufficient storage space is available to perform the operation

Panic Reasons

- object is not a valid handle on a persistent object opened with the write access right.

5.8.4 TEE_SeekObjectData

```
TEE_Result TEE_SeekObjectData(
           TEE_ObjectHandle object,
           int32_t   offset,
           TEE_Whence whence )
```

Description

The TEE_SeekObjectData function sets the data position indicator associated with the object handle.

The parameter whence controls the meaning of offset:

- If whence is TEE_DATA_SEEK_SET, the data position is set to offset bytes from the beginning of the data stream.
- If whence is TEE_DATA_SEEK_CUR, the data position is set to its current position plus offset.
- If whence is TEE_DATA_SEEK_END, the data position is set to the size of the object data plus offset.

The TEE_SeekObjectData function may be used to set the data position beyond the end of stream; this does not constitute an error. However, the data position indicator does have a maximum value which is TEE_DATA_MAX_POSITION. If the value of the data position indicator resulting from this operation would be greater than TEE_DATA_MAX_POSITION, the error TEE_ERROR_OVERFLOW is returned.

If an attempt is made to move the data position before the beginning of the data stream, the data position is set at the beginning of the stream. This does not constitute an error.

Parameters

- object: The object handle
- offset: The number of bytes to move the data position. A positive value moves the data position forward; a negative value moves the data position backward.
- whence: The position in the data stream from which to calculate the new position

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_OVERFLOW: If the value of the data position indicator resulting from this operation would be greater than TEE_DATA_MAX_POSITION

Panic Reasons

- object is not a valid handle on a persistent object.

6 Cryptographic Operations API

This part of the Cryptographic API defines how to actually perform cryptographic operations:

- Cryptographic operations can be pre-allocated for a given operation type, algorithm, and key size. Resulting ***Cryptographic Operation Handles*** can be reused for multiple operations.
- When required by the operation, the ***Cryptographic Operation Key*** can be set up independently and reused for multiple operations. Note that some cryptographic algorithms, such as AES-XTS, require two keys.
- The cryptographic algorithms listed in Table 6-1 are supported in this specification.

Table 6-1: Supported Cryptographic Algorithms

Digests	MD5 SHA-1 SHA-256 SHA-224 SHA-384 SHA-512
Symmetric ciphers	DES Triple-DES with double-length and triple-length keys AES
Message Authentication Codes (MACs)	DES-MAC AES-MAC AES-CMAC HMAC with one of the supported digests
Authenticated Encryption (AE)	AES-CCM with support for Additional Authenticated Data (AAD) AES-GCM with support for Additional Authenticated Data (AAD)
Asymmetric Encryption Schemes	RSA PKCS1-V1.5 RSA OAEP
Asymmetric Signature Schemes	DSA RSA PKCS1-V1.5 RSA PSS
Key Exchange Algorithms	Diffie-Hellman

- Digest, symmetric ciphers, MACs, and AE operations are always multi-stage, i.e., data can be provided in successive chunks to the API. On the other hand, asymmetric operations are always single stage. Note that signature and verification operations operate on a digest computed by the caller.
- Operation states can be copied from one operation handle into an uninitialized operation handle. This allows the TA to duplicate or fork a multi-stage operation, for example.

6.1 Data Types

6.1.1 TEE_OperationMode

The enumeration TEE_OperationMode lists the modes for all the cryptographic operations.

```
typedef enum {
   TEE_MODE_ENCRYPT,
   TEE_MODE_DECRYPT
   TEE_MODE_SIGN,
   TEE_MODE_VERIFY,
   TEE_MODE_MAC,
   TEE_MODE_DIGEST,
   TEE_MODE_DERIVE
} TEE_OperationMode;
```

Table 6-2: Possible TEE_OperationMode Values

Name	Comment
TEE_MODE_ENCRYPT	Encryption mode
TEE_MODE_DECRYPT	Decryption mode
TEE_MODE_SIGN	Signature generation mode
TEE_MODE_VERIFY	Signature verification mode
TEE_MODE_MAC	MAC mode
TEE_MODE_DIGEST	Digest mode
TEE_MODE_DERIVE	Key derivation mode

6.1.2 TEE_OperationInfo

```
typedef struct {
   uint32_t algorithm;
   uint32_t operationClass;
   uint32_t mode;
   uint32_t digestLength;
   uint32_t maxKeySize;
   uint32_t keySize;
   uint32_t requiredKeyUsage;
   uint32_t handleState;
} TEE_OperationInfo;
```

See the documentation of function TEE_GetOperationInfo in section 6.2.3 for a description of this structure.

6.1.3 TEE_OperationHandle

```
typedef struct __TEE_OperationHandle* TEE_OperationHandle
```

TEE_OperationHandle is an opaque handle on a cryptographic operation. These handles are returned by the function TEE_AllocateOperation specified in section 6.2.1.

6.2 Generic Operation Functions

These functions are common to all the types of cryptographic operations, which are:

- Digests
- Symmetric ciphers
- MACs
- Authenticated Encryptions
- Asymmetric operations
- Key Derivations

6.2.1 TEE_AllocateOperation

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

Description

The `TEE_AllocateOperation` function allocates a handle for a new cryptographic operation and sets the mode and algorithm type. If this function does not return with `TEE_SUCCESS` then there is no valid handle value.

Once a cryptographic operation has been created, the implementation MUST guarantee that all resources necessary for the operation are allocated and that any operation with a key of at most `maxKeySize` bits can be performed.

The parameter `algorithm` must be one of the constants defined in section 6.10.1.

The parameter `mode` must be one of the constants defined in section 6.1.1. It must be compatible with the algorithm as defined by Table 6-3.

Table 6-3: TEE_AllocateOperation: Allowed Modes

Algorithm	Possible Modes
TEE_ALG_AES_ECB_NOPAD	TEE_MODE_ENCRYPT TEE_MODE_DECRYPT
TEE_ALG_AES_CBC_NOPAD	
TEE_ALG_AES_CTR	
TEE_ALG_AES_CTS	
TEE_ALG_AES_XTS	
TEE_ALG_AES_CCM	
TEE_ALG_AES_GCM	
TEE_ALG_DES_ECB_NOPAD	
TEE_ALG_DES_CBC_NOPAD	
TEE_ALG_DES3_ECB_NOPAD	
TEE_ALG_DES3_CBC_NOPAD	
TEE_ALG_DES_CBC_MAC_NOPAD	TEE_MODE_MAC
TEE_ALG_AES_CBC_MAC_NOPAD	
TEE_ALG_AES_CBC_MAC_PKCS5	
TEE_ALG_AES_CMAC	
TEE_ALG_DES_CBC_MAC_PKCS5	
TEE_ALG_DES3_CBC_MAC_NOPAD	
TEE_ALG_DES3_CBC_MAC_PKCS5	
TEE_ALG_RSASSA_PKCS1_V1_5_MD5	TEE_MODE_SIGN TEE_MODE_VERIFY
TEE_ALG_RSASSA_PKCS1_V1_5_SHA1	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA224	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA256	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA384	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA512	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512	
TEE_ALG_DSA_SHA1	

Algorithm	Possible Modes
TEE_ALG_RSAES_PKCS1_V1_5	TEE_MODE_ENCRYPT TEE_MODE_DECRYPT
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512	
TEE_ALG_RSA_NOPAD	
TEE_ALG_DH_DERIVE_SHARED_SECRET	TEE_MODE_DERIVE
TEE_ALG_MD5	TEE_MODE_DIGEST
TEE_ALG_SHA1	
TEE_ALG_SHA224	
TEE_ALG_SHA256	
TEE_ALG_SHA384	
TEE_ALG_SHA512	
TEE_ALG_HMAC_MD5	TEE_MODE_MAC
TEE_ALG_HMAC_SHA1	
TEE_ALG_HMAC_SHA224	
TEE_ALG_HMAC_SHA256	
TEE_ALG_HMAC_SHA384	
TEE_ALG_HMAC_SHA512	

Note that all algorithms listed in Table 6-3 must be supported by any compliant Implementation, but a particular implementation may also support more implementation-defined algorithms, modes, or key sizes.

Parameters

- `operation`: Reference to generated operation handle
- `algorithm`: One of the cipher algorithms enumerated in section 6.1.1
- `mode`: The operation mode
- `maxKeySize`: Maximum key size in bits for the operation

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_OUT_OF_MEMORY`: If there are not enough resources to allocate the operation
- `TEE_ERROR_NOT_SUPPORTED`: If the mode is not compatible with the algorithm or key size or if the algorithm is not one of the listed algorithms

6.2.2 TEE_FreeOperation

```
void TEE_FreeOperation(
   TEE_OperationHandle operation)
```

Description

The `TEE_Free Operation` function deallocates all resources associated with an operation handle. After this function is called, the operation handle is no longer valid.

This function cannot fail.

Parameters

- `operation`: Reference to operation handle

Panic Reasons

- `operation` is not a valid handle on an operation.

6.2.3 TEE_GetOperationInfo

```
void TEE_GetOperationInfo(
          TEE_OperationHandle operation,
   [out] TEE_OperationInfo*  operationInfo
)
```

Description

The TEE_GetOperationInfo function returns information about an operation handle. It fills the following fields in the structure operationInfo (defined in section 6.1.2):

- algorithm, mode, maxKeySize: The parameters passed to the function TEE_AllocateOperation
- algorithmClass: One of the following constants, describing the kind of operation:
  - TEE_OPERATION_CIPHER
  - TEE_OPERATION_MAC
  - TEE_OPERATION_AE
  - TEE_OPERATION_DIGEST
  - TEE_OPERATION_ASYMMETRIC_CIPHER
  - TEE_OPERATION_ASYMMETRIC_SIGNATURE
  - TEE_OPERATION_KEY_DERIVATION
- keySize: If a key is programmed in the operation, the actual size of this key
- requiredKeyUsage: A bit vector that describes the necessary bits in the object usage for TEE_SetOperationKey or TEE_SetOperationKey2 to succeed without panicking. Set to 0 for a digest operation.
- digestLength: For a MAC, AE, or Digest digest, describes the number of bytes in the digest or tag
- handleState: A bit vector describing the current state of the operation. Contains one or more of the following flags:
  - TEE_HANDLE_FLAG_EXPECT_TWO_KEYS: Set if the algorithm expects two keys to be set, using TEE_SetOperationKey2. This happens only if algorithm is set to TEE_ALG_AES_XTS.
  - TEE_HANDLE_FLAG_KEY_SET: Set if the operation key has been set. Always set for digest operations.
  - TEE_HANDLE_FLAG_INITIALIZED: For multi-stage operations, i.e., all but TEE_OPERATION_ASYMMETRIC_XXX operation classes, whether the operation has been initialized using one of the TEE_XXXInit functions.

Parameters

- operation: Handle on the operation
- operationInfo: Pointer to a structure filled with the operation information

Panic Reasons

- operation is not a valid opened operation handle.

6.2.4 TEE_ResetOperation

```
void TEE_ResetOperation(
   TEE_OperationHandle operation,
)
```

Description

For a multi-stage operation, the `TEE_ResetOperation` function resets the operation state before initialization, but after the key has been set.

This function can be called on any operation and at any time, but is meaningful only for the multi-stage operations, i.e., symmetric ciphers, MACs, AEs, and digests.

When such a multi-stage operation is active, i.e., when it has been initialized but not yet successfully finalized, then the operation is reset to its pre-initialization state. The operation key(s) are not cleared.

Note that it is valid to call the initialization functions while the operation is active. In this case, the operation is first reset, then initialized.

Parameters

- `operation`: Handle on the operation

Panic Reasons

- `operation` is not a valid opened operation handle.

6.2.5 TEE_SetOperationKey

```
TEE_Result TEE_SetOperationKey(
   TEE_OperationHandle operation,
   TEE_ObjectHandle key
)
```

Description

The TEE_SetOperationKey function programs the key of an operation; that is, it associates an operation with a key.

The key material is **copied** from the key object handle into the operation. After the key has been set, there is no longer any link between the operation and the key object. The object handle can be closed or reset and this will not affect the operation.

This function accepts handles on both transient key objects and persistent key objects.

The key object type and size must be compatible with the type and size of the operation. The operation mode must be compatible with key usage:

- In general, the operation mode must be allowed in the object usage.
- For the TEE_ALG_RSA_NOPAD algorithm:
  - The only supported modes are TEE_MODE_ENCRYPT and TEE_MODE_DECRYPT.
  - For TEE_MODE_ENCRYPT, the object usage must contain both the TEE_USAGE_ENCRYPT and TEE_USAGE_VERIFY flags.
  - For TEE_MODE_DECRYPT, the object usage must contain both the TEE_USAGE_DECRYPT and TEE_USAGE_SIGN flags.
- For a public key object, the operation mode can only be TEE_MODE_ENCRYPT or TEE_MODE_VERIFY but cannot be TEE_MODE_DECRYPT or TEE_MODE_SIGN as these kinds of operations require the private parts of a key-pair.
- If the object is a key-pair then the key parts used in the operation depend on the mode:
  - For a TEE_MODE_ENCRYPT or TEE_MODE_VERIFY operation, the public parts of the key-pair are used.
  - For a TEE_MODE_DECRYPT or TEE_MODE_SIGN, the private parts of the key-pair are used.

If key is set to TEE_HANDLE_NULL, then the operation key is cleared.

Parameters

- operation: Operation handle
- key: A handle on a key object

Return Value

The only possible return value is TEE_SUCCESS. The presence of an error return value is for future versions of the specification.

Panic Reasons

- `operation` is not a valid opened operation handle.
- `key` is not `TEE_HANDLE_NULL` and is not a valid handle on a key object.
- `key` is not initialized.
- The operation expects no key (digest mode) or two keys (AES-XTS algorithm) .
- The type, size, or usage of `key` is not compatible with the algorithm, mode, or size of the `operation`.

6.2.6 TEE_SetOperationKey2

```
TEE_Result TEE_SetOperationKey2(
   TEE_OperationHandle operation,
   TEE_ObjectHandle key1,
   TEE_ObjectHandle key2
)
```

Description

The TEE_SetOperationKey2 function initializes an existing operation with two keys. This is used only for the algorithm TEE_ALG_AES_XTS.

This function works like TEE_SetOperationKey except that two keys are set instead of a single key.

key1 and key2 must be both non-NULL or both NULL.

Parameters

- operation: Operation handle
- key1: A handle on a key object
- key2: A handle on a key object

Return Value

The only possible return value is TEE_SUCCESS. The presence of an error return value is for future versions of the specification.

Panic Reasons

- operation is not a valid opened operation handle.
- key1 and key2 are not both TEE_HANDLE_NULL and key1 or key2 or both are not valid handles on a key object.
- key1 and/or key2 are not initialized.
- The operation expects no key (digest mode) or a single key (all but AES-XTS algorithm) .
- The type, size, or usage of key1 or key2 is not compatible with the algorithm, mode, or size of the operation.

6.2.7 TEE_CopyOperation

```
void TEE_CopyOperation(
   TEE_OperationHandle dstOperation,
   TEE_OperationHandle srcOperation
)
```

Description

The `TEE_CopyOperation` function copies an operation state from one operation handle into another operation handle. This also copies the key material associated with the source operation.

The state of `srcOperation` including the key material currently set up is copied into `dstOperation`.

This function is useful in the following use cases:

- "Forking" a digest operation after feeding some amount of initial data
- Computing intermediate digests

The algorithm and mode of `dstOperation` must be equal to the algorithm and mode of `srcOperation`.

If `srcOperation` has no key programmed, then the key in `destOperation` is cleared. If there is a key programmed in `srcOperation`, then the maximum key size of `dstOperation` can be greater than or equal to the actual key size of `srcOperation`.

Parameters

- `dstOperation`: Handle on the destination operation
- `srcOperation`: Handle on the source operation

Panic Reasons

- `dstOperation` or `srcOperation` is not a valid opened operation handle.
- The algorithm and mode differ in `dstOperation` and `srcOperation`.
- `srcOperation` has a key and its size is greater than the maximum key size of `dstOperation`.

6.3 Message Digest Functions

6.3.1 TEE_DigestUpdate

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

Description

The `TEE_DigestUpdate` function accumulates message data for hashing. The message does not have to be block aligned. Subsequent calls to this function are possible.

Parameters

- `operation`: Handle of a running Message Digest operation
- `chunk`, `chunkSize`: Chunk of data to be hashed

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_DIGEST`.

6.3.2 TEE_DigestDoFinal

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

Description

The `TEE_DigestDoFinal` function finalizes the message digest operation and produces the message hash. Afterwards the Message Digest operation is reset and can be reused.

Parameters

- `operation`: Handle of a running Message Digest operation
- `chunk`, `chunkLen`: Last chunk of data to be hashed
- `hash`, `hashLen`: Output buffer filled with the message hash

Return Value

- `TEE_SUCCESS`: On success
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is too small. In this case, the operation is not finalized.

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_DIGEST`.

6.4 Symmetric Cipher Functions

These functions define the way to perform symmetric cipher operations, such as AES. They cover both block ciphers and stream ciphers.

6.4.1 TEE_CipherInit

```
void TEE_CipherInit(
            TEE_OperationHandle operation,
   [inbuf]  void* IV, size_t IVLen
)
```

Description

The `TEE_CipherInit` function starts the symmetric cipher operation.

The operation must have been associated with a key.

If the operation is already active, it is reset, then initialized.

This function cannot fail.

Parameters

- `operation`: A handle on an opened cipher operation setup with a key
- `IV`, `IVLen`: Buffer containing the operation Initialization Vector

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_CIPHER`.
- No key is programmed in the `operation`.
- The Initialization Vector does not have the length required by the algorithm.

6.4.2 TEE_CipherUpdate

```
TEE_Result TEE_CipherUpdate(
            TEE_OperationHandle operation,
   [inbuf]  void* srcData,  size_t srcLen,
   [outbuf] void* destData, size_t *destLen
)
```

Description

The `TEE_CipherUpdate` function encrypts or decrypts input data.

Input data does not have to be a multiple of block size. Subsequent calls to this function are possible. Unless one or more calls of this function have supplied sufficient input data, no output is generated. The cipher operation is finalized with a call to `TEE_CipherDoFinal`.

Parameters

- `operation`: Handle of a running Cipher operation
- `srcData`, `srcLen`: Input data buffer to be encrypted or decrypted
- `destData`, `destLen`: Output buffer

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to contain the output. In this case, the input is not fed into the algorithm.

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_CIPHER`.
- The operation has not been started yet with `TEE_CipherInit` or has already been finalized.

6.4.3 TEE_CipherDoFinal

```
TEE_Result TEE_CipherDoFinal(
               TEE_OperationHandle operation,
   [inbuf]     void* srcData,  size_t srcLen,
   [outbufopt] void* destData, size_t *destLen
)
```

Description

The `TEE_CipherDoFinal` function finalizes the cipher operation, processing data that has not been processed by previous calls to `TEE_CipherUpdate` as well as data supplied in `srcData`. The operation handle can be reused or newly initialized.

Parameters

- `operation`: Handle of a running Cipher operation
- `srcData`, `srcLen`: Reference to final chunk of input data to be encrypted or decrypted
- `destData`, `destLen`: Output buffer. Can be omitted if the output is to be discarded, e.g., because it is known to be empty.

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to contain the output

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_CIPHER`.
- The operation has not been started yet with `TEE_CipherInit` or has already been finalized.

6.5 MAC Functions

These functions are used to perform MAC (Message Authentication Code) operations, such as HMAC or AES-CMAC operations.

These functions are not used for Authenticated Encryption algorithms, which must use the functions defined in section 6.6.

6.5.1 TEE_MACInit

```
void TEE_MACInit(
            TEE_OperationHandle operation,
   [inbuf]  void* IV, size_t IVLen
)
```

Description

The `TEE_MACInit` function initializes a MAC operation.

The operation is started and associated with a key. If this function does not return with TEE_SUCCESS, the operation is not initialized.

If the MAC algorithm does not require an IV, the parameters `IV`, `IVLen` are ignored. `IV` must still be a valid input buffer, so, for example, if `IV` is `NULL`, then `IVLen` must be set to `0`.

Parameters

- `operation`: Operation handle
- `IV`, `IVLen`: Input buffer containing the operation IV, if applicable

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- No key is programmed in the `operation`.
- The Initialization Vector does not have the length required by the algorithm.

6.5.2 TEE_MACUpdate

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

Description

The `TEE_MACUpdate` function accumulates data for a MAC calculation.

Input data does not have to be a multiple of the block size. Subsequent calls to this function are possible. `TEE_MACComputeFinal` or `TEE_MACCompareFinal` are called to complete the MAC operation.

Parameters

- `operation`: Handle of a running MAC operation
- `chunk`, `chunkSize`: Chunk of the message to be MACed

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- The operation has not been started yet with `TEE_MACInit` or has already been finalized.

6.5.3 TEE_MACComputeFinal

```
TEE_Result TEE_MACComputeFinal(
            TEE_OperationHandle operation,
   [inbuf]  void* message, size_t messageLen,
   [outbuf] void* mac, size_t *macLen
)
```

Description

The `TEE_MACComputeFinal` function finalizes the MAC operation with a last chunk of message, and computes the MAC. Afterwards the operation handle can be reused and initialized with a new key.

Parameters

- `operation`: Handle of a MAC operation
- `message`, `messageLen`: Input buffer containing a last message chunk to MAC
- `mac`, `macLen`: Output buffer filled with the computed MAC

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to contain the computed MAC

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- The operation has not been started yet with `TEE_MACInit` or has already been finalized.

6.5.4 TEE_MACCompareFinal

```
TEE_Result TEE_MACCompareFinal(
            TEE_OperationHandle operation,
   [inbuf]  void* message, size_t messageLen,
   [inbuf]  void* mac, size_t *macLen
)
```

Description

The `TEE_MACCompareFinal` function finalizes the MAC operation and compares the MAC with the buffer passed to the function. Afterwards the operation handle can be reused and initialized with a new key.

Parameters

- `operation`: Handle of a MAC operation
- `message`, `messageLen`: Input buffer containing the last message chunk to MAC
- `mac`, `macLen`: Input buffer containing the MAC to check

Return Value

- `TEE_SUCCESS`: If the computed MAC corresponds to the MAC passed in the parameter `mac`
- `TEE_ERROR_MAC_INVALID`: If the computed MAC does not correspond to the value passed in the parameter `mac`

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- The operation has not been started yet with `TEE_MACInit` or has already been finalized.

6.6 Authenticated Encryption Functions

These functions are used for Authenticated Encryption operations, i.e., the TEE_ALG_AES_CCM and TEE_ALG_AES_GCM algorithms.

6.6.1 TEE_AEInit

```
TEE_Result TEE_AEInit(
           TEE_OperationHandle operation,
   [inbuf] void*      nonce, size_t nonceLen,
           uint32_t   tagLen,
           uint32_t   AADLen,
           uint32_t   payloadLen
)
```

Description

The TEE_AEInit function initializes an Authentication Encryption operation.

Parameters

- operation: A handle on the operation
- nonce, nonceLen: The operation nonce or IV
- tagLen: Size in bits of the tag length
  - For AES-GCM, can be 128, 120, 112, 104, or 96
  - For AES-CCM, can be 128, 112, 96, 64, 48, or 32
- AADLen: Length in bytes of the AAD
  - Used only for AES-CCM. Ignored for AES-GCM.
- payloadLen: Length in bytes of the payload
  - Used only for AES-CCM. Ignored for AES-GCM.

Return Value

- TEE_SUCCESS: On success
- TEE_ERROR_NOT_SUPPORTED: If the tag length is not supported by the algorithm

Panic Reasons

- operation is not a valid operation handle of class TEE_OPERATION_AE.
- No key is programmed in the operation.
- The nonce length is not compatible with the length required by the algorithm.

6.6.2 TEE_AEUpdateAAD

```
void TEE_AEUpdateAAD(
           TEE_OperationHandle operation,
   [inbuf] void* AADdata, size_t AADdataLen
)
```

Description

The TEE_AEUpdateAAD function feeds a new chunk of Additional Authentication Data (AAD) to the AE operation.

Parameters

- operation: Handle on the AE operation
- AADdata, AADdataLen: Input buffer containing the chunk of AAD

Panic Reasons

- operation is not a valid operation handle of class TEE_OPERATION_AE.
- The operation has not started yet.
- The AAD length has already been reached.

6.6.3 TEE_AEUpdate

```
TEE_Result TEE_AEUpdate(
            TEE_OperationHandle operation,
   [inbuf]  void* srcData,  size_t srcLen,
   [outbuf] void* destData, size_t *destLen
)
```

Description

The TEE_AEUpdate function accumulates data for an Authentication Encryption operation.

Input data does not have to be a multiple of block size. Subsequent calls to this function are possible. Unless one or more calls of this function have supplied sufficient input data, no output is generated. The AE operation is finalized with a call to TEE_AEDoFinal.

Parameters

- operation: Handle of a running AE operation
- srcData, srcLen: Input data buffer to be encrypted or decrypted
- destData, destLen: Output buffer

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_SHORT_BUFFER: If the output buffer is not large enough to contain the output

Panic Reasons

- operation is not a valid operation handle of class TEE_OPERATION_AE.
- The operation has not started yet.
- The required AAD length has not been provided yet.
- The payload length has already been reached.

6.6.4 TEE_AEEncryptFinal

```
TEE_Result TEE_AEEncryptFinal(
               TEE_OperationHandle operation,
   [inbuf]     void* srcData,  size_t   srcLen,
   [outbuf]    void* destData, size_t*  destLen,
   [outbuf]    void* tag,      size_t* tagLen
)
```

Description

The `TEE_AEEncryptFinal` function processes data that has not been processed by previous calls to `TEE_AEUpdate` as well as data supplied in `srcData`. It completes the AE operation and computes the tag.

The operation handle can be reused or newly initialized.

Parameters

- `operation`: Handle of a running AE operation
- `srcData`, `srcLen`: Reference to final chunk of input data to be encrypted
- `destData`, `destLen`: Output buffer. Can be omitted if the output is to be discarded, e.g., because it is known to be empty.
- `tag`, `tagLen`: Output buffer filled with the computed tag

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to contain the output

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_AE`.
- The operation has not started yet.
- The required AAD and payload have not been provided.

6.6.5 TEE_AEDecryptFinal

```
TEE_Result TEE_AEDecryptFinal(
                TEE_OperationHandle operation,
   [inbuf]      void* srcData, size_t srcLen,
   [outbuf]     void* destData, size_t *destLen,
   [in]         void* tag, size_t tagLen
)
```

Description

The `TEE_AEDecryptFinal` function processes data that has not been processed by previous calls to `TEE_AEUpdate` as well as data supplied in `srcData`. It completes the AE operation and compares the computed tag with the tag supplied in the parameter `tag`.

The operation handle can be reused or newly initialized.

Parameters

- `operation`: Handle of a running AE operation
- `srcData`, `srcLen`: Reference to final chunk of input data to be decrypted
- `destData`, `destLen`: Output buffer. Can be omitted if the output is to be discarded, e.g., because it is known to be empty.
- `tag`, `tagLen`: Input buffer containing the tag to compare

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to contain the output
- `TEE_ERROR_MAC_INVALID`: If the computed tag does not match the supplied tag

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_AE`.
- The operation has not started yet.
- The required AAD and payload have not been provided.

6.7 Asymmetric Functions

These functions allow the encryption and decryption of data using asymmetric algorithms, signatures of digests, and verification of signatures.

Note that asymmetric encryption is always “single-stage”, which differs from symmetric ciphers which are always “multi-stage”.

6.7.1 TEE_AsymmetricEncrypt, TEE_AsymmetricDecrypt

```
TEE_Result TEE_AsymmetricEncrypt(
                TEE_OperationHandle operation,
   [in]         TEE_Attribute*      params,    uint32_t paramCount
   [inbuf]      void*               srcData,   size_t   srcLen,
   [outbuf]     void*               destData,  size_t  *destLen,
)
TEE_Result TEE_AsymmetricDecrypt(
                TEE_OperationHandle operation,
   [in]         TEE_Attribute*     params,     uint32_t paramCount
   [inbuf]      void*               srcData,    size_t   srcLen,
   [outbuf]     void*               destData,   size_t  *destLen
)
```

Description

The TEE_AsymmetricEncrypt function encrypts a message within an asymmetric operation, and the TEE_AsymmetricDecrypt function decrypts the result.

These functions can be called only with an operation of the following algorithms:

- TEE_ALG_RSAES_PKCS1_V1_5
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512
- TEE_ALG_RSA_NOPAD

The parameters params, paramCount contain the operation parameters listed in Table 6-4.

Table 6-4: Asymmetric Encrypt/Decrypt Operation Parameters

Algorithm	Possible operation parameters
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_XXX	TEE_ATTR_RSA_OAEP_LABEL: This parameter is optional. If not present, an empty label is assumed.

Parameters

- `operation`: Handle on the operation, which must have been suitably set up with an operation key
- `params`, `paramCount`: Optional operation parameters
- `srcData`, `srcLen`: Input buffer
- `destData`, `destLen`: Output buffer

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to hold the result

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_ASYMMETRIC_CIPHER`.
- No key is programmed in the operation.
- The mode is not compatible with the function.

6.7.2 TEE_AsymmetricSignDigest

```
TEE_Result TEE_AsymmetricSignDigest(
              TEE_OperationHandle operation,
   [in]       TEE_Attribute*      params,     uint32_t paramCount
   [inbuf]    void*               digest,     size_t   digestLen,
   [outbuf]   void*               signature,  size_t  *signatureLen
)
```

Description

The `TEE_AsymmetricSignDigest` function signs a message digest within an asymmetric operation.

Note that only an already hashed message can be signed.

This function can be called only with an operation of the following algorithms:

- `TEE_ALG_RSASSA_PKCS1_V1_5_MD5`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA1`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA224`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA256`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA384`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA512`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512`
- `TEE_ALG_DSA_SHA1`

The parameters `params`, `paramCount` contain the operation parameters listed in Table 6-5.

Table 6-5: Asymmetric Sign Operation Parameters

Algorithm	Possible operation parameters
`TEE_ALG_RSASSA_PKCS1_PSS_MGF1_XXX`	`TEE_ATTR_RSA_PSS_SALT_LENGTH`: Number of bytes in the salt. This parameter is optional. If not present, the salt length is equal to the hash length.

Parameters

- `operation`: Handle on the operation, which must have been suitably set up with an operation key
- `params`, `paramCount`: Optional operation parameters
- `digest`, `digestLen`: Input buffer containing the input message digest
- `signature`, `signatureLen`: Output buffer written with the signature of the digest

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_SHORT_BUFFER: If the output buffer is not large enough to hold the result

Panic Reasons

- operation is not a valid operation handle of class TEE_OPERATION_ASYMMETRIC_SIGNATURE.
- No key is programmed in the operation.
- The operation mode is not TEE_MODE_SIGN.

6.7.3 TEE_AsymmetricVerifyDigest

```
TEE_Result TEE_AsymmetricVerifyDigest(
              TEE_OperationHandle operation,
   [in]       TEE_Attribute*      params,    uint32_t paramCount,
   [inbuf]    void*               digest,    size_t  digestLen,
   [inbuf]    void*               signature, size_t  signatureLen
)
```

Description

The `TEE_AsymmetricVerifyDigest` function verifies a message digest signature within an asymmetric operation.

This function can be called only with an operation of the following algorithms:

- `TEE_ALG_RSASSA_PKCS1_V1_5_MD5`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA1`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA224`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA256`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA384`
- `TEE_ALG_RSASSA_PKCS1_V1_5_SHA512`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384`
- `TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512`
- `TEE_ALG_DSA_SHA1`

The parameters `params`, `paramCount` contain the operation parameters listed in Table 6-6.

Table 6-6: Asymmetric Verify Operation Parameters

Algorithm	Possible operation parameters
`TEE_ALG_RSASSA_PKCS1_PSS_MGF1_XXX`	`TEE_ATTR_RSA_PSS_SALT_LENGTH`: Number of bytes in the salt. This parameter is optional. If not present, the salt length is equal to the hash length.

Parameters

- `operation`: Handle on the operation, which must have been suitably set up with an operation key
- `params`, `paramCount`: Optional operation parameters
- `digest`, `digestLen`: Input buffer containing the input message digest
- `signature`, `signatureLen`: Input buffer containing the signature to verify

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_SIGNATURE_INVALID: If the signature is invalid

Panic Reasons

- operation is not a valid operation handle of class TEE_OPERATION_ASYMMETRIC_SIGNATURE.
- No key is programmed in the operation.
- The operation mode is not TEE_MODE_VERIFY.

6.8 Key Derivation Functions

6.8.1 TEE_DeriveKey

```
void TEE_DeriveKey(
          TEE_OperationHandle operation,
   [in]   TEE_Attribute*      params, uint32_t paramCount
          TEE_ObjectHandle    derivedKey
)
```

Description

The `TEE_DeriveKey` function can only be used with the algorithm `TEE_ALG_DH_DERIVE_SHARED_SECRET`.

The parameters `params`, `paramCount` contain the operation parameters listed in Table 6-7.

Table 6-7: Asymmetric Derivation Operation Parameters

Algorithm	Possible operation parameters
`TEE_ALG_DH_DERIVE_SHARED_SECRET`	`TEE_ATTR_DH_PUBLIC_VALUE`: Public key part of the other party. This parameter is mandatory. The `derivedKey` handle must refer to an object with type `TEE_TYPE_GENERIC_SECRET`.

Parameters

- `operation`: Handle on the operation, which must have been suitably set up with an operation key
- `params`, `paramCount`: Operation parameters
- `derivedKey`: Handle on an uninitialized transient object filled with the derived key

Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_KEY_DERIVATION`.
- No key is programmed in the operation.
- A mandatory parameter is missing.
- The operation mode is not `TEE_MODE_DERIVE`.

6.9 Random Data Generation Function

6.9.1 TEE_GenerateRandom

```
void TEE_GenerateRandom(
   [out] void*       randomBuffer,
         size_t      randomBufferLen
)
```

Description

The TEE_GenerateRandom function generates random data.

Parameters

- randomBuffer: Reference to generated random data
- randomBufferLen: Byte length of requested random data

6.10 Cryptographic Algorithms Specification

This section specifies the cryptographic algorithms, key types, and key parts supported in the Cryptographic Operations API.

Note that for the "NOPAD" symmetric algorithms, it is the responsibility of the TA to do the paddings.

6.10.1 List of Algorithm Identifiers

Table 6-8 provides an exhaustive list of all algorithm identifiers specified in the Cryptographic Operations API.

Table 6-8: List of Algorithm Identifiers

Name	Identifier	Comments
TEE_ALG_AES_ECB_NOPAD	0x10000010	
TEE_ALG_AES_CBC_NOPAD	0x10000110	
TEE_ALG_AES_CTR	0x10000210	The counter MUST be encoded as a 16-byte buffer in big-endian form. Between two consecutive blocks, the counter MUST be incremented by 1. If it reaches the limit of all 128 bits set to 1, it MUST wrap around to 0.
TEE_ALG_AES_CTS	0x10000310	
TEE_ALG_AES_XTS	0x10000410	
TEE_ALG_AES_CBC_MAC_NOPAD	0x30000110	
TEE_ALG_AES_CBC_MAC_PKCS5	0x30000510	
TEE_ALG_AES_CMAC	0x30000610	
TEE_ALG_AES_CCM	0x40000710	
TEE_ALG_AES_GCM	0x40000810	
TEE_ALG_DES_ECB_NOPAD	0x10000011	
TEE_ALG_DES_CBC_NOPAD	0x10000111	
TEE_ALG_DES_CBC_MAC_NOPAD	0x30000111	
TEE_ALG_DES_CBC_MAC_PKCS5	0x30000511	
TEE_ALG_DES3_ECB_NOPAD	0x10000013	Triple DES must be understood as Encrypt-Decrypt-Encrypt mode with two or three keys.
TEE_ALG_DES3_CBC_NOPAD	0x10000113	
TEE_ALG_DES3_CBC_MAC_NOPAD	0x30000113	
TEE_ALG_DES3_CBC_MAC_PKCS5	0x30000513	
TEE_ALG_RSASSA_PKCS1_V1_5_MD5	0x70001830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA1	0x70002830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA224	0x70003830	

Name	Identifier	Comments
TEE_ALG_RSASSA_PKCS1_V1_5_SHA256	0x70004830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA384	0x70005830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA512	0x70006830	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1	0x70212930	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224	0x70313930	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256	0x70414930	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384	0x70515930	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512	0x70616930	
TEE_ALG_RSAES_PKCS1_V1_5	0x60000130	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1	0x60210230	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224	0x60210230	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256	0x60210230	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384	0x60210230	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512	0x60210230	
TEE_ALG_RSA_NOPAD	0x60000030	
TEE_ALG_DSA_SHA1	0x70002131	
TEE_ALG_DH_DERIVE_SHARED_SECRET	0x80000032	
TEE_ALG_MD5	0x50000001	
TEE_ALG_SHA1	0x50000002	
TEE_ALG_SHA224	0x50000003	
TEE_ALG_SHA256	0x50000004	
TEE_ALG_SHA384	0x50000005	
TEE_ALG_SHA512	0x50000006	
TEE_ALG_HMAC_MD5	0x30000001	
TEE_ALG_HMAC_SHA1	0x30000002	
TEE_ALG_HMAC_SHA224	0x30000003	
TEE_ALG_HMAC_SHA256	0x30000004	
TEE_ALG_HMAC_SHA384	0x30000005	
TEE_ALG_HMAC_SHA512	0x30000006	

Implementer's Note

Note that the algorithm identifiers have the structure defined in Table 6-9.

Table 6-9: Structure of Algorithm Identifier

Bits	Function	Values
Bits [31:28]	Specify the algorithm class and determine which function can be called	0x1: Block cipher 0x3: MAC 0x4: Authenticated Encryption cipher 0x5: Digest 0x6: Asymmetric cipher 0x7: Asymmetric signature 0x8: Key derivation
Bits [7:0]	Identify the underlying main algorithm itself	0x01: MD5 0x02: SHA-1 0x03: SHA-224 0x04: SHA-256 0x05: SHA-384 0x06: SHA-512 0x10: AES 0x11: DES 0x12: DES2 (only for key generation) 0x13: DES3 0x30: RSA 0x31: DSA 0x32: DH
Bits [11:8]	Define the chaining mode or padding	
Bits [15:12]	Define the message digest for asymmetric signature algorithms	
Bits [19:16]	Define the MGF for RSA PSS and RSA OAEP algorithms	
Bits [23:20]	Define the internal hash used by the MGF for RSA OAEP (for signature algorithms, equal to the message digest)	
Bits [27:24]	Not used	

6.10.2 Object Types

Table 6-10: List of Object Types

Name	Identifier	Possible sizes
TEE_TYPE_AES	0xA0000010	128, 192, 256 bits
TEE_TYPE_DES	0xA0000011	56 bits
TEE_TYPE_DES3	0xA0000013	112 and 168 bits
TEE_TYPE_HMAC_MD5	0xA0000001	
TEE_TYPE_HMAC_SHA1	0xA0000002	
TEE_TYPE_HMAC_SHA224	0xA0000003	
TEE_TYPE_HMAC_SHA256	0xA0000004	
TEE_TYPE_HMAC_SHA384	0xA0000005	
TEE_TYPE_HMAC_SHA512	0xA0000006	
TEE_TYPE_RSA_PUBLIC_KEY	0xA0000030	
TEE_TYPE_RSA_KEYPAIR	0xA1000030	
TEE_TYPE_DSA_PUBLIC_KEY	0xA0000031	
TEE_TYPE_DSA_KEYPAIR	0xA1000031	
TEE_TYPE_DH_KEYPAIR	0xA1000032	
TEE_TYPE_GENERIC_SECRET	0xA0000000	

6.11 Object or Operation Attributes

Table 6-11: Object or Operation Attributes

Name	Value	Protection	Type	Comment
TEE_ATTR_SECRET_VALUE	0xC0000000	Protected	Ref	Used for all secret keys for symmetric ciphers, MACs, and HMACs
TEE_ATTR_RSA_MODULUS	0xD0000130	Public	Ref	
TEE_ATTR_RSA_PUBLIC_EXPONENT	0xD0000230	Public	Ref	
TEE_ATTR_RSA_PRIVATE_EXPONENT	0xC0000330	Protected	Ref	
TEE_ATTR_RSA_PRIME1	0xC0000430	Protected	Ref	This is usually referred to as *p*.
TEE_ATTR_RSA_PRIME2	0xC0000530	Protected	Ref	*q*
TEE_ATTR_RSA_EXPONENT1	0xC0000630	Protected	Ref	*dp*
TEE_ATTR_RSA_EXPONENT2	0xC0000730	Protected	Ref	*dq*
TEE_ATTR_RSA_COEFFICIENT	0xC0000830	Protected	Ref	*iq*
TEE_ATTR_DSA_PRIME	0xD0001031	Public	Ref	*p*
TEE_ATTR_DSA_SUBPRIME	0xD0001131	Public	Ref	*q*
TEE_ATTR_DSA_BASE	0xD0001231	Public	Ref	*g*
TEE_ATTR_DSA_PUBLIC_VALUE	0xD0000131	Public	Ref	*y*
TEE_ATTR_DSA_PRIVATE_VALUE	0xC0000231	Protected	Ref	*x*
TEE_ATTR_DH_PRIME	0xD0001032	Public	Ref	*p*
TEE_ATTR_DH_SUBPRIME	0xD0001132	Public	Ref	*q*
TEE_ATTR_DH_BASE	0xD0001232	Public	Ref	*g*
TEE_ATTR_DH_X_BITS	0xF0001332	Public	Value	ℓ
TEE_ATTR_DH_PUBLIC_VALUE	0xD0000132	Public	Ref	*y*
TEE_ATTR_DH_PRIVATE_VALUE	0xC0000232	Protected	Ref	*x*
TEE_ATTR_RSA_OAEP_LABEL	0xD0000930	Public	Ref	
TEE_ATTR_RSA_PSS_SALT_LENGTH	0xF0000A30	Public	Value	

Implementer's Notes

Selected bits of the attribute identifiers are explained in Table 6-12.

Table 6-12: Partial Structure of Attribute Identifier

Bit	Function	Values
Bit [29]	Defines whether the attribute is a buffer or value attribute	0: buffer attribute 1: value attribute
Bit [28]	Defines whether the attribute is protected or public	0: protected attribute 1: public attribute

A protected attribute cannot be extracted unless the object has the TEE_USAGE_EXTRACTABLE flag.

Table 6-13 defines constants that reflect setting bit [29] and bit [28], respectively, intended to help decode attribute identifiers.

Table 6-13: Attribute Identifier Flags

Name	Value
TEE_ATTR_FLAG_VALUE	0x20000000
TEE_ATTR_FLAG_PUBLIC	0x10000000

7 Time API

This API provides access to three sources of time:

- ***System Time***
  - The origin of this system time is arbitrary and implementation-dependent. Different TA instances may even have different system times. The only guarantee is that the system time is not reset or rolled back during the life of a given TA instance, so it can be used to compute time differences and operation deadlines, for example. The system time must not be affected by transitions through low power states.
  - System time is related to the function `TEE_Wait`, which waits for a given timeout or cancellation.
  - The level of trust that a Trusted Application can put on the system time is implementation defined but can be discovered programmatically by querying the implementation property "`gpd.tee.systemTime.protectionLevel`". Typically, an implementation may rely on the REE timer (protection level 100) or on a dedicated secure timer hardware (protection level 1000).
- ***TA Persistent Time***, a real-time source of time
  - The origin of this time is set individually by each Trusted Application and must persist across reboots.
  - The level of trust on the TA Persistent Time can be queried through the property "`gpd.tee.TAPersistentTime.protectionLevel`".
- ***REE Time***
  - This is as trusted as the REE itself and may also be tampered by the user.

All time functions use a millisecond resolution and split the time in the two fields of the structure `TEE_Time`: one field for the seconds and one field for the milliseconds within this second.

7.1 Data Types

7.1.1 TEE_Time

```
typedef struct
{
   uint32_t seconds;
   uint32_t millis;
}
TEE_Time;
```

7.2 Time Functions

7.2.1 TEE_GetSystemTime

```
void TEE_GetSystemTime(
   [out] TEE_Time* time
)
```

Description

The TEE_GetSystemTime function retrieves the current system time.

The system time has an arbitrary implementation-defined origin that can vary across TA instances. The minimum guarantee is that the system time must be monotonous for a given TA instance.

Implementations are allowed to use the REE timers to implement this function but may also better protect the system time. A TA can discover the level of protection implementation by querying the implementation property gpd.tee.systemTime.protectionLevel. Possible values are listed in Table 7-1.

Table 7-1: Values of the gpd.tee.systemTime.protectionLevel Property

Value	Meaning
100	System time based on REE-controlled timers. Can be tampered by the REE. The implementation must still guarantee that the system time is monotonous, i.e., successive calls to TEE_GetSystemTime must return increasing values of the system time.
1000	System time based on a TEE-controlled secure timer. The REE cannot interfere with the system time. It may still interfere with the scheduling of TEE tasks, but is not able to hide delays from a TA calling TEE_GetSystemTime.

7.2.2 TEE_Wait

```
TEE_Result TEE_Wait(
   uint32_t timeout
)
```

Description

The TEE_Wait function waits for the specified number of milliseconds or waits forever if timeout equals TEE_TIMEOUT_INFINITE (0xFFFFFFFF).

When this function returns success, the implementation must guarantee that the difference between two calls to TEE_GetSystemTime before and after the call to TEE_Wait is greater than or equal to the requested timeout. However, there may be additional implementation-dependent delays due to the scheduling of TEE tasks.

This function is cancellable, i.e., if the current task's cancelled flag is set and the TA has unmasked the effects of cancellation, then this function returns earlier than the requested timeout with the error code TEE_ERROR_CANCEL. See section 4.10 for more details about cancellations.

Return Value

- TEE_SUCCESS: On success
- TEE_ERROR_CANCEL: If the wait has been cancelled

7.2.3 TEE_GetTAPersistentTime

```
TEE_Result TEE_GetTAPersistentTime(
        [out] TEE_Time* time)
```

Description

The TEE_GetTAPersistentTime function retrieves the persistent time of the Trusted Application, expressed as a number of seconds and milliseconds since the arbitrary origin set by calling TEE_SetTAPersistentTime.

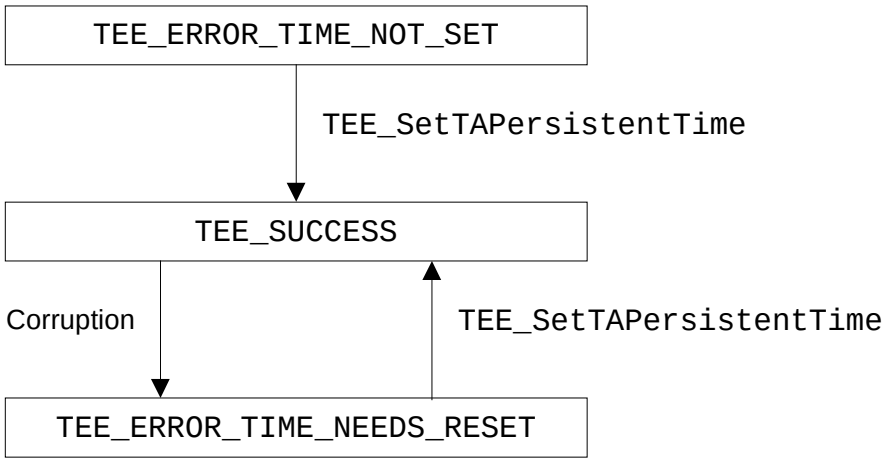
This function can return the following statuses (as well as other status values discussed in "Return Value"):

- TEE_SUCCESS means the persistent time is correctly set and has been retrieved into the parameter time.
- TEE_ERROR_TIME_NOT_SET is the initial status and means the persistent time has not been set. The Trusted Application must set its persistent time by calling the function TEE_SetTAPersistentTime.
- TEE_ERROR_TIME_NEEDS_RESET means the persistent time has been set but may have been corrupted and must no longer be trusted. In such a case it is recommended that the Trusted Application resynchronize the persistent time by calling the function TEE_SetTAPersistentTime. Until the persistent time has been reset, the status TEE_ERROR_TIME_NEEDS_RESET will be always returned.

Initially the persistent time status is TEE_ERROR_TIME_NOT_SET. Once a Trusted Application has synchronized its persistent time by calling TEE_SetTAPersistentTime, the status can be TEE_SUCCESS or TEE_ERROR_TIME_NEEDS_RESET. Once the status has become TEE_ERROR_TIME_NEEDS_RESET it will keep this status until the persistent time is re-synchronized by calling TEE_SetTAPersistentTime.

Figure 7-1 shows the state machine of the persistent time status.

Figure 7-1: Persistent Time Status State Machine



The meaning of the status TEE_ERROR_TIME_NEEDS_RESET depends on the protection level provided by the hardware implementation and the underlying real-time clock (RTC). This protection level can be queried by retrieving the implementation property gpd.tee.TAPersistentTime.protectionLevel, which can have one of the values listed in Table 7-2.

Table 7-2: Values of the `gpd.tee.TAPersistentTime.protectionLevel` Property

Value	Meaning
100	Persistent time based on an REE-controlled real-time clock and on the TEE Trusted Storage for the storage of origins. The implementation must guarantee that rollback of persistent time is detected to the fullest extent allowed by the Trusted Storage.
1000	Persistent time based on a TEE-controlled real-time clock and the TEE Trusted Storage. The real-time clock must be out of reach of software attacks from the REE. Users may still be able to provoke a reset of the real-time clock but this must be detected by the Implementation.

The number of seconds in the TA Persistent Time may exceed the range of the `uint32_t` integer type. In this case, the function MUST return the error `TEE_ERROR_OVERFLOW`, but still computes the TA Persistent Time as specified above, except that the number of seconds is truncated to 32 bits before being written to `time->seconds`. For example, if the Trusted Application sets its persistent time to 2^{32}-100 seconds, then after 100 seconds, the TA Persistent Time is 2^{32}, which is not representable with a `uint32_t`. In this case, the function `TEE_GetPersistentTime` MUST return `TEE_ERROR_OVERFLOW` and set `time->seconds` to `0` (which is 2^{32} truncated to 32 bits).

Parameters

- `time`: A pointer to the placeholder to be set with the current TA Persistent Time. For an error different from `TEE_ERROR_OVERFLOW`, this placeholder is filled with zeros.

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_TIME_NOT_SET`
- `TEE_ERROR_TIME_NEEDS_RESET`
- `TEE_ERROR_OVERFLOW`: The number of seconds in the TA Persistent Time overflows the range of a `uint32_t`. The field `time->seconds` is still set to the TA Persistent Time truncated to 32 bits (i.e., modulo 2^{32}).
- `TEE_ERROR_OUT_OF_MEMORY`: If not enough memory is available to complete the operation

7.2.4 TEE_SetTAPersistentTime

```
TEE_Result TEE_SetTAPersistentTime(
        [in] TEE_Time* time)
```

Description

The TEE_SetTAPersistentTime function sets the persistent time of the current Trusted Application.

Only the persistent time for the current Trusted Application is modified, not the persistent time of other Trusted Applications. This will affect all the instances of the current Trusted Application. The modification is atomic and persistent across device reboots.

Parameters

- time: Filled with the persistent time of the current TA

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_OUT_OF_MEMORY: If not enough memory is available to complete the operation
- TEE_ERROR_STORAGE_NO_SPACE: If insufficient storage space is available to complete the operation

7.2.5 TEE_GetREETime

```
void TEE_GetREETime(
   [out] TEE_Time* time
)
```

Description

The TEE_GetREETime function retrieves the current REE system time. This function retrieves the current time as seen from the point of view of the REE, expressed in the number of seconds since midnight on January 1, 1970, UTC.

In normal operation, the value returned should correspond to the real time, but it should not be considered as trusted, as it may be tampered by the user or the REE software.

Parameter

- time: Filled with the number of seconds and milliseconds since midnight on January 1, 1970, UTC

8 TEE Arithmetical API

8.1 Introduction

All asymmetric cryptographic functions are implemented by using arithmetical functions, where operands are typically elements of finite fields or in mathematical structures containing finite field elements. The Cryptographic Operations API should hide the complexity of the mathematics that is behind these operations. A developer who needs some cryptographic service does not need to know anything about the internal implementation.

However in practice developer may face the following difficulties: the API does not support the desired algorithm; or the API supports the algorithm, but with the wrong encodings, options, etc. The purpose of the TEE Arithmetical API is to provide building blocks so that the developer can implement missing asymmetric algorithms. In other words the arithmetical API can be used to implement a plug-in into the Cryptographic Operations API. Allowing the possibility of expanding the Cryptographic Operations API means that some of its functions can be left as optional to implement.

Furthermore and to ease the design of speed efficient algorithms, the arithmetical API also gives access to a Fast Modular Multiplication primitive, referred to as FMM.

This specification mandates that all functions within the TEE Arithmetical API MUST work when input and output `TEE_BigInt` values are within the interval [-2^M+1, 2^M-1] (limits included), where M is an implementation-defined number of bits. Every Implementation MUST ensure that M is at least 2048. The exact value of M can be retrieved as the implementation property `gpd.tee.arith.maxBigIntSize`.

Throughout this chapter:

- The notation “n-bit integer” refers to an integer that can take values in the range [-2^n+1, 2^n-1], including limits.
- The notation “`magnitude(src)`” denotes the minimum number of required bits to represent the absolute value of the big integer `src` in a natural binary representation. The developer may query the magnitude of a big integer by using the function `TEE_BigIntGetBitCount(src)`, as described in section 8.7.5.

8.2 Error Handling and Parameter Checking

This low level arithmetical API performs very few checks on the parameters given to the functions. Most functions will return undefined results when called inappropriately but will not generate any error return values.

Some functions in the API MAY work for inputs larger than indicated by the implementation property `gpd.tee.arith.maxBigIntSize`. This is however not guaranteed. When a function does not support a given bigint size beyond this limit, it MUST panic and not produce invalid results.

8.3 Data Types

This specification version has three data types for the arithmetical operations. These are `TEE_BigInt`, `TEE_BigIntFMM`, and `TEE_BigIntFMMContext`. Before using the arithmetic operations, the TA developer must allocate and initialize the memory for the input and output operands This API provides entry points to determine the correct sizes of the needed memory allocations.

8.3.1 TEE_BigInt

The `TEE_BigInt` type is a placeholder for the memory structure of the TEE core internal representation of a large multi-precision integer.

```
typedef uint32_t TEE_BigInt;
```

The following constraints are put on the internal representation of the `TEE_BigInt`:

1) The size of the representation must be a multiple of 4 bytes.

2) The extra memory within the representation to store metadata must not exceed 8 bytes.

3) The representation must be stored 32-bit aligned in memory.

Exactly how a multi-precision integer is represented internally is implementation specific but it must be stored within a structure of the maximum size given by the macro `TEE_BigIntSizeInU32` (see section 8.4.1).

By defining a `TEE_BigInt` as a `uint32_t` for the TA, we allow the TA developer to allocate static space for multiple occurrences of `TEE_BigInt` at compile time which obey constraints 1 and 3. The allocation can be done with code similar to this:

```
uint32_t     twoints[2 * TEE_BigIntSizeInU32(1024)];
TEE_BigInt*  first  = twoints;
TEE_BigInt*  second = twoints + TEE_BigIntSizeInU32(1024);

/* Or if we do it dynamically */
TEE_BigInt* op1;
op1  = TEE_Malloc(TEE_BigIntSizeInU32(1024) * sizeof(TEE_BigInt));
/* use op1 */
TEE_Free(op1);
```

Conversions from an external representation to the internal `TEE_BigInt` representation and vice versa can be done by using functions from section 8.6.

Most functions in the TEE Arithmetical API take one or more `TEE_BigInt` pointers as parameters; for example, `func(TEE_BigInt *op1, TEE_BigInt *op2)`. When describing the parameters and what the function does, this specification will refer to the integer represented in the structure to which the pointer `op1` points, by simply writing `op1`. It will be clear from the context when the pointer value is referred to and when the integer value is referred to.

8.3.2 TEE_BigIntFMMContext

Usually, such a fast modular multiplication requires some additional data or derived numbers. That extra data is stored in a context that must be passed to the fast modular multiplication function. The TEE_BigIntFMMContext is a placeholder for the TEE core internal representation of the context that is used in the fast modular multiplication operation.

```
typedef uint32_t TEE_BigIntFMMContext;
```

The following constraints are put on the internal representation of the TEE_BigIntFMMContext:

1) The size of the representation must be a multiple of 4 bytes.

2) The representation must be stored 32-bit aligned in memory.

Exactly how this context is represented internally is implementation specific but it must be stored within a structure of the size given by the function TEE_BigIntFMMContextSizeInU32 (see section 8.4.2).

Similarly to TEE_BigInt, we expose this type as a uint32_t to the TA, which helps TEE_Malloc to align the structure correctly when allocating space for a TEE_BigIntFMMContext*.

8.3.3 TEE_BigIntFMM

Some implementations may have support for faster modular multiplication algorithms such as Montgomery or Barrett multiplication for use in modular exponentiation. Typically, those algorithms require some transformation of the input before the multiplication can be carried out. The TEE_BigIntFMM is a placeholder for the memory structure that holds an integer in such a transformed representation.

```
typedef uint32_t TEE_BigIntFMM;
```

The following constraints are put on the internal representation of the TEE_BigIntFMM:

1) The size of the representation must be a multiple of 4 bytes.

2) The representation must be stored 32-bit aligned in memory.

Exactly how this transformed representation is stored internally is implementation specific but it must be stored within a structure of the maximum size given by the function TEE_BigIntFMMSizeInU32 (see section 8.4.2).

Similarly to TEE_BigInt, we expose this type as a uint32_t to the TA, which helps TEE_Malloc to align the structure correctly when allocating space for a TEE_BigIntFMM*.

8.4 Memory Allocation and Size of Objects

It is the responsibility of the Trusted Application to allocate and free memory for all TEE arithmetical objects, including all operation contexts, used in the Trusted Application. Once the arithmetical objects are allocated, the functions in the TEE Arithmetical API will never fail because of out-of-resources.

TEE implementer's note: Implementations of the TEE Arithmetical API should utilize memory from one or more pre-allocated pools to store intermediate results during computations to ensure that the functions does not fail because of out-of-resources problems. All memory resources used internally MUST be thread-safe to avoid re-entrance issues. Such a pool of scratch memory could be:

- Internal memory of a hardware accelerator module
- Allocated from mutex protected system-wide memory
- Allocated from the heap of the TA instance, i.e., by using `TEE_Malloc` or `TEE_Realloc`

If the implementation uses a memory pool of temporary storage for intermediate results or if it uses hardware resources such as accelerators for some computations, the implementation MUST either wait for the resource to become available or, for example in case of a busy hardware accelerator, resort to other means such as a software implementation.

8.4.1 TEE_BigIntSizeInU32

```
#define TEE_BigIntSizeInU32(n) ((((n)+31)/32)+2)
```

Description

The `TEE_BigIntSizeInU32` macro calculates the size of the array of `uint32_t` values needed to represent an n-bit integer. This is defined as a macro (thereby mandating the maximum size of the internal representation) rather than as a function so that TA developers can use the macro in a static compile-time declaration of an array. Note that the implementation of the internal arithmetic functions assumes that memory pointed to by the `TEE_BigInt*` is 32-bit aligned.

Parameters

- n: maximum number of bits to be representable

8.4.2 TEE_BigIntFMMContextSizeInU32

```
size_t TEE_BigIntFMMContextSizeInU32(
     size_t    modulusSizeInBits
)
```

Description

The `TEE_BigIntFMMContextSizeInU32` function returns the size of the array of `uint32_t` values needed to represent a fast modular context using a given modulus size. This function MUST never fail.

Parameters

- `modulusSizeInBits`: Size of modulus in bits

Return Value

Number of bytes needed to store a `TEE_BigIntFMMContext` given a modulus of length `modulusSizeInBits`

8.4.3 TEE_BigIntFMMSizeInU32

```
size_t TEE_BigIntFMMSizeInU32(
    size_t    modulusSizeInBits
)
```

Description

The TEE_BigIntFMMSizeInU32 function returns the size of the array of uint32_t values needed to represent an integer in the fast modular multiplication representation, given the size of the modulus in bits. This function MUST never fail.

Normally from a mathematical point of view, this function would have needed the context to compute the exact required size. However, it is beneficial to have a function that does not take an initialized context as a parameter and thus the implementation may overstate the required memory size. It is nevertheless likely that a given implementation of the fast modular multiplication can calculate a very reasonable upper-bound estimate based on the modulus size.

Parameters

- modulusSizeInBits: Size of modulus in bits

Return Value

Number of bytes needed to store a TEE_BigIntFMM given a modulus of length modulusSizeInBits

8.5 Initialization Functions

These functions initialize the arithmetical objects after the TA has allocated the memory to store them. The Trusted Application MUST call the corresponding initialization function after it has allocated the memory for the arithmetical object.

8.5.1 TEE_BigIntInit

```
void TEE_BigIntInit(
   [out]   TEE_BigInt  *bigInt,
           size_t       len
)
```

Description

The `TEE_BigIntInit function` initializes `bigInt` and sets its represented value to zero. This function assumes that `bigInt` points to a memory area of `len uint32_t`. This can be done for example with the following memory allocation:

```
TEE_BigInt *a;
size_t len;
len = TEE_BigIntSizeInU32(bitSize);
a =(TEE_BigInt *)TEE_Malloc(len * sizeof(TEE_BigInt));
ret = TEE_BigIntInit(a, len);
```

Parameters

- `bigInt`: A pointer to the `TEE_BigInt` to be initialized
- `len`: The size in `uint32_t` of the memory pointed to by `bigInt`

8.5.2 TEE_BigIntInitFMMContext

```
void TEE_BigIntInitFMMContext(
   [out]   TEE_BigIntFMMContext   *context,
           size_t                 len,
   [in]    TEE_BigInt             *modulus
)
```

Description

The `TEE_BigIntInitFMMContext` function calculates the necessary prerequisites for the fast modular multiplication and stores them in a context. This function assumes that `context` points to a memory area of `len` `uint32_t`. This can be done for example with the following memory allocation:

```
TEE_BigIntFMMContext* ctx;
size_t len = TEE_BigIntFMMContextSizeInU32(bitsize);
ctx=(TEE_BigIntFMMContext *)TEE_Malloc(len * sizeof(TEE_BigIntFFMContext));
/*Code for initializing modulus*/
…
TEE_BigIntInitFMMContext(ctx, len, modulus);
```

Even though a fast multiplication might be mathematically defined for any modulus, normally there are restrictions in order for it to be fast on a computer. This specification mandates that all implementations MUST work for all odd moduli larger than 2 and less than 2 to the power of the implementation defined property `gpd.tee.arith.maxBigIntSize`.

Parameters

- `context`: A pointer to the `TEE_BigIntFMMContext` to be initialized
- `len`: The size in `uint32_t` of the memory pointed to by `context`
- `modulus`: The modulus, an odd integer larger than 2 and less than 2 to the power of `gpd.tee.arith.maxBigIntSize`

8.5.3 TEE_BigIntInitFMM

```
void TEE_BigIntInitFMM(
   [in]    TEE_BigIntFMM     *bigIntFMM,
           size_t            len
)
```

Description

The TEE_BigIntInitFMM function initializes bigIntFMM and sets its represented value to zero. This function assumes that bigIntFMM points to a memory area of len uint32_t. This can be done for example with the following memory allocation:

```
TEE_BigIntFMM *a;
size_t len;
len = TEE_BigIntFMMSizeInU32(modulusSizeinBits);
a = (TEE_BigIntFMM *)TEE_Malloc(len * sizeof(TEE_BigIntFMM));
TEE_InitFMMInt(a, len);
```

Parameters

- bigIntFMM: A pointer to the TEE_BigIntFMM to be initialized
- len: The size in uint32_t of the memory pointed to by bigIntFMM

8.6 Converter Functions

TEE_BigInt contains the internal representation of a multi-precision integer. However in many use cases some integer data comes from external sources or integers; for example, a local device gets an ephemeral Diffie-Hellman public key during a key agreement procedure. In this case the ephemeral key is expected to be in octet string format, which is a big endian radix 256 representation for unsigned numbers. For example 0x123456789abcdef has the following octet string representation:

```
{0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef}
```

This section provides functions to convert to and from such alternative representations.

8.6.1 TEE_BigIntConvertFromOctetString

```
TEE_Result TEE_BigIntConvertFromOctetString(
   [out]    TEE_BigInt *dest,
   [inbuf]  uint8_t    *buffer, size_t bufferLen,
            int32_t     sign
)
```

Description

The TEE_BigIntConvertFromOctetString function converts a bufferLen byte octet string buffer into a TEE_BigInt format. The octet string is in most significant byte first representation. The input parameter sign will set the sign of dest. It will be set to negative if sign<0 and to positive if sign>=0.

Parameters

- dest: Pointer to a TEE_BigInt to hold the result
- buffer: Pointer to the buffer containing the octet string representation of the integer
- bufferLen: The length of *buffer in bytes
- sign: The sign of dest is set to the sign of sign.

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_OVERFLOW: If memory allocation for the dest is too small

8.6.2 TEE_BigIntConvertToOctetString

```
TEE_Result TEE_BigIntConvertToOctetString(
   [outbuf]    void *buffer, size_t *bufferLen,
   [in]        TEE_BigInt   *bigInt
)
```

Description

The `TEE_BigIntConvertToOctetString` function converts the absolute value of an integer in `TEE_BigInt` format into an octet string. The octet string is written in a most significant byte first representation.

Parameters

- `buffer`, `bufferLen`: Output buffer where converted octet string representation of the integer is written
- `bigInt`: Pointer to the integer that will be converted to an octet string

Return Value

- `TEE_SUCCESS`: In case of success
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is too small to contain the octet string

8.6.3 TEE_BigIntConvertFromS32

```
void TEE_BigIntConvertFromS32(
   [out]   TEE_BigInt  *dest,
           int32_t     shortVal
)
```

Description

The TEE_BigIntConvertFromS32 function sets *dest to the value shortVal.

Parameters

- dest: Pointer to a TEE_BigInt to store the result
- shortVal: Input value

Result Size

The result must have memory allocation for holding a 32-bit signed value.

8.6.4 TEE_BigIntConvertToS32

```
TEE_Result TEE_BigIntConvertToS32(
   [out]   int32_t     *dest,
   [in]    TEE_BigInt  *src
)
```

Description

The TEE_BigIntConvertToS32 function sets *dest to the value of src, including the sign of src. If src does not fit within an int32_t, the value of *dest is undefined.

Parameters

- dest: Pointer to an int32_t to store the result
- src: Pointer to the input value

Return Value

- TEE_SUCCESS: In case of success
- TEE_ERROR_OVERFLOW: If src does not fit within an int32_t

8.7 Logical Operations

8.7.1 TEE_BigIntCmp

```
int32_t TEE_BigIntCmp(
     [in]  TEE_BigInt *op1,
     [in]  TEE_BigInt *op2
)
```

Description

The TEE_BigIntCmp function checks whether op1>op2, op1==op2, or op1<op2.

Parameters

- op1: Pointer to the first operand
- op2: Pointer to the second operand

Return Value

This function returns a negative number if op1<op2, 0 if op1==op2, and a positive number if op1>op2.

8.7.2 TEE_BigIntCmpS32

```
int32_t TEE_BigIntCmpS32(
   [in]    TEE_BigInt  *op,
           int32_t     shortVal
)
```

Description

The TEE_BigIntCmpS32 function checks whether op>shortVal, op==shortVal, or op<shortVal.

Parameters

- op: Pointer to the first operand
- shortVal: Pointer to the second operand

Return Value

This function returns a negative number if op<shortVal, 0 if op==shortVal, and a positive number if op>shortVal.

8.7.3 TEE_BigIntShiftRight

```
void TEE_BigIntShiftRight(
   [out]   TEE_BigInt  *dest,
   [in]    TEE_BigInt  *op
           size_t      bits
)
```

Description

The `TEE_BigIntShiftRight` function computes `|dest| = |op| >> bits` and `dest` will have the same sign as `op`.[4] If `bits` is greater than the bit length of `op` then the result is zero. It is allowed that `dest` and `op` point to the same memory region.

Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the shifted result
- `op`: Pointer to the operand to be shifted
- `bits`: Number of bits to shift

8.7.4 TEE_BigIntGetBit

```
bool TEE_BigIntGetBit(
     [in]  TEE_BigInt  *src,
           uint32_t    bitIndex
)
```

Description

The `TEE_BigIntGetBit` function returns the `bitIndex`th bit of the natural binary representation of `|src|`. A true return value indicates a “1” and a false return value indicates a “0” in the `bitIndex`th position. If `bitIndex` is larger than the number of bits in `op`, the return value is false, thus indicating a “0”.

Parameters

- `src`: Pointer to the integer
- `bitIndex`: The offset of the bit to be read, starting at offset 0 for the least significant bit

Return Value

The Boolean value of the `bitIndex`th bit in `|src|`. True represents a “1” and false represents a “0”.

[4] The notation `|x|` means the absolute value of `x`.

8.7.5 TEE_BigIntGetBitCount

```
uint32_t TEE_BigIntGetBitCount(
     [in]  TEE_BigInt  *src
)
```

Description

The TEE_BigIntGetBitCount function returns the number of bits in the natural binary representation of |src|; that is, the magnitude of src.

Parameters

- src: Pointer to the integer

Return Value

The number of bits in the natural binary representation of |src|. If src equals zero, it will return 0.

8.8 Basic Arithmetic Operations

This section describes basic arithmetical operations addition, subtraction, negation, multiplication, squaring, and division.

8.8.1 TEE_BigIntAdd

```
void TEE_BigIntAdd(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op1,
   [in]    TEE_BigInt *op2
)
```

Description

The `TEE_BigIntAdd` function computes `dest` = `op1` + `op2`. It is allowed that all or some of `dest`, `op1`, and `op2` point to the same memory region.

Parameters

- `dest`: Pointer to `TEE_BigInt` to store the result `op1` + `op2`
- `op1`: Pointer to the first operand
- `op2`: Pointer to the second operand

Result Size

Depending on the sign of `op1` and `op2`, the result may be larger or smaller than `op1` and `op2`. For the worst case, `dest` must have memory allocation for holding `max(magnitude(op1), magnitude(op2))+1` bits.[5]

[5] The magnitude function is defined in section 8.7.5.

8.8.2 TEE_BigIntSub

```
void TEE_BigIntSub(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op1,
   [in]    TEE_BigInt *op2
)
```

Description

The TEE_BigIntSub function computes dest = op1 – op2. It is allowed that all or some of dest, op1, and op2 point to the same memory region.

Parameters

- dest: Pointer to TEE_BigInt to store the result op1 - op2
- op1: Pointer to the first operand
- op2: Pointer to the second operand

Result Size

Depending on the sign of op1 and op2, the result may be larger or smaller than op1 and op2. For the worst case, the result must have memory allocation for holding max(magnitude(op1), magnitude(op2))+1 bits.

8.8.3 TEE_BigIntNeg

```
void TEE_BigIntNeg(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op
)
```

Description

The TEE_BigIntNeg function negates an operand: dest = -op. It is allowed that dest and op point to the same memory region.

Parameters

- dest: Pointer to TEE_BigInt to store the result -op
- op: Pointer to the operand to be negated

Result Size

The result must have memory allocation for magnitude(op) bits.

8.8.4 TEE_BigIntMul

```
void TEE_BigIntMul(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op1,
   [in]    TEE_BigInt *op2
)
```

Description

The `TEE_BigIntMul` function computes `dest = op1 * op2`. It is allowed that all or some of `dest`, `op1`, and `op2` point to the same memory region.

Parameters

- `dest`: Pointer to `TEE_BigInt` to store the result `op1 * op2`
- `op1`: Pointer to the first operand
- `op2`: Pointer to the second operand

Result Size

The result must have memory allocation for `(magnitude(op1) + magnitude(op2))` bits.

8.8.5 TEE_BigIntSquare

```
void TEE_BigIntSquare(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op,
)
```

Description

The `TEE_BigIntSquare` function computes `dest = op * op`. It is allowed that `dest` and `op` point to the same memory region.

Parameters

- `dest`: Pointer to `TEE_BigInt` to store the result `op * op`
- `op`: Pointer to the operand to be squared

Result Size

The result must have memory allocation for `2*magnitude(op)` bits.

8.8.6 TEE_BigIntDiv

```
void TEE_BigIntDiv(
   [out]   TEE_BigInt *dest_q,
   [out]   TEE_BigInt *dest_r,
   [in]    TEE_BigInt *op1
   [in]    TEE_BigInt *op2
)
```

Description

The TEE_BigIntDiv function computes dest_r and dest_q such that op1 = dest_q * op2 + dest_r. It will round dest_q towards zero and dest_r will have the same sign as op1.

Example:

op1	op2	dest_q	dest_r	Expression
53	7	7	4	53 = 7*7 + 4
-53	7	-7	-4	-53 = (-7)*7 + (-4)
53	-7	-7	+4	53 = (-7)*(-7) + 4
-53	-7	7	-4	-53 = 7*(-7) + (-4)

If op2 equals zero this function will execute a divide-by-zero operation using the normal / operator for integers.

This function has the limitation that the memory pointed to by dest_q and dest_r MUST NOT overlap. However it is possible that dest_q==op1, dest_q==op2, dest_r==op1, dest_r==op2, when dest_q and dest_r do not overlap. It is allowed to pass a NULL pointer for either dest_q or dest_r in which case the implementation MAY take advantage of the fact that it is only required to calculate either dest_q or dest_r.

Parameters

- dest_q: Pointer to a TEE_BigInt to store the quotient. dest_q can be NULL.
- dest_r: Pointer to a TEE_BigInt to store the remainder. dest_r can be NULL.
- op1: Pointer to the first operand, the dividend
- op2: Pointer to the second operand, the divisor

Result Sizes

The quotient, dest_q, must have memory allocation for 0 bytes if |op1| <= |op2| and magnitude(op1) – magnitude(op2) if |op1| > |op2|.

The remainder dest_r must have memory allocation to hold magnitude(op2) bits.

8.9 Modular Arithmetic Operations

To reduce the number of tests the modular functions needs to perform on entrance and to speed up the performance, all modular functions assume that input operands are normalized, i.e., non-negative and smaller than the modulus, and the modulus should be greater than one, otherwise it is a Programmer Error and the behavior of these functions are undefined. This normalization can be done by using the reduction function in section 8.9.1.

8.9.1 TEE_BigIntMod

```
void TEE_BigIntMod(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op,
   [in]    TEE_BigInt *n
)
```

Description

The `TEE_BigIntMod` function computes `dest` = `op (mod n)` such that `0` <= `dest` < `n`. It is allowed that `dest` and `op` point to the same memory region but `n` MUST point to a unique memory region. For negative `op` it follows the normal convention that `-1` = `(n-1) mod n`.

Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result `op (mod n)`. The result `dest` will be in the interval `[0, n-1]`.
- `op`: Pointer to the operand to be reduced `mod n`
- `n`: Pointer to the modulus. Modulus must be larger than 1.

Result Size

The result `dest` must have memory allocation for `magnitude(n)` bits.[6]

Panic Reasons

- `n<2`

[6] The magnitude function is defined in section 8.7.5.

8.9.2 TEE_BigIntAddMod

```
void TEE_BigIntAddMod(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op1,
   [in]    TEE_BigInt *op2
   [in]    TEE_BigInt *n
)
```

Description

The `TEE_BigIntAddMod` function computes `dest` = `op1` + `op2` (`mod` `n`). It is allowed that all or some of `dest`, `op1`, and `op2` point to the same memory region but `n` MUST point to a unique memory region.

Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result `op1` + `op2` (`mod` `n`)
- `op1`: Pointer to the first operand. Operand must be in the interval `[0,n-1]`.
- `op2`: Pointer to the second operand. Operand must be in the interval `[0,n-1]`.
- `n`: Pointer to the modulus. Modulus must be larger than 1.

Result Size

The result `dest` must have memory allocation for `magnitude(n)` bits.

Panic Reasons

- `n<2`

8.9.3 TEE_BigIntSubMod

```
void TEE_BigIntSubMod(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op1,
   [in]    TEE_BigInt *op2
   [in]    TEE_BigInt *n
)
```

Description

The `TEE_BigIntSubMod` function computes `dest` = `op1 - op2 (mod n)`. It is allowed that all or some of `dest`, `op1`, and `op2` points to the same memory region but n MUST point to a unique memory region.

Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result `op1 - op2 (mod n)`
- `op1`: Pointer to the first operand. Operand must be in the interval `[0,n-1]` .
- `op2`: Pointer to the second operand. Operand must be in the interval `[0,n-1]` .
- `n`: Pointer to the modulus. Modulus must be larger than 1.

Result Size

The result `dest` must have memory allocation for `magnitude(n)` bits.

Panic Reasons

- `n<2`

8.9.4 TEE_BigIntMulMod

```
void TEE_BigIntMulMod(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op1,
   [in]    TEE_BigInt *op2
   [in]    TEE_BigInt *n
)
```

Description

The `TEE_BigIntMulMod` function computes `dest = op1 * op2 (mod n)`. It is allowed that all or some of `dest`, `op1`, and `op2` point to the same memory region but `n` MUST point to a unique memory region.

Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result `op1 * op2 (mod n)`
- `op1`: Pointer to the first operand. Operand must be in the interval `[0,n-1]`.
- `op2`: Pointer to the second operand. Operand must be in the interval `[0,n-1]`.
- `n`: Pointer to the modulus. Modulus must be larger than 1.

Result Size

The result `dest` must have memory allocation for `magnitude(n)` bits.

Panic Reasons

- `n<2`

8.9.5 TEE_BigIntSquareMod

```
void TEE_BigIntSquareMod(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op,
   [in]    TEE_BigInt *n
)
```

Description

The `TEE_BigIntSquareMod` function computes `dest = op * op (mod n)`. It is allowed that `dest` and `op1` point to the same memory region but `n` MUST point to a unique memory region.

Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result `op * op (mod n)`
- `op`: Pointer to the operand. Operand must be in the interval `[0,n-1]`.
- `n`: Pointer to the modulus. Modulus must be larger than 1.

Result Size

The result `dest` must have memory allocation for `magnitude(n)` bits.

Panic Reasons

- `n<2`

8.9.6 TEE_BigIntInvMod

```
void TEE_BigIntInvMod(
   [out]   TEE_BigInt *dest,
   [in]    TEE_BigInt *op,
   [in]    TEE_BigInt *n
)
```

Description

The `TEE_BigIntInvMod` function computes `dest` such that `dest * op = 1 (mod n)`. It is allowed that `dest` and `op` point to the same memory region. This function assumes that `gcd(op,n)` is equal to `1`, which can be checked by using the function in section 8.10.1. If `gcd(op,n)` is greater than `1` then the result is unreliable.

Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result `op^-1 (mod n)`
- `op`: Pointer to the operand. Operand must be in the interval `[1,n-1]`.
- `n`: Pointer to the modulus. Modulus must be larger than 1.

Result Size

The result `dest` must have memory allocation for `magnitude(n)` bits.

Panic Reasons

- `n<2`
- `op = 0`

8.10 Other Arithmetic Operations

8.10.1 TEE_BigIntRelativePrime

```
bool TEE_BigIntRelativePrime(
   [in]    TEE_BigInt *op1,
   [in]    TEE_BigInt *op2
)
```

Description

The `TEE_BigIntRelativePrime` function determines whether `gcd(op1, op2)==1`. It is allowed that `op1` and `op2` point to the same memory region.

Parameters

- `op1`: Pointer to the first operand
- `op2`: Pointer to the second operand

Return Value

- `true` if `gcd(op1, op2)==1`
- `false` otherwise

8.10.2 TEE_BigIntComputeExtendedGcd

```
void TEE_BigIntComputeExtendedGcd(
   [out]   TEE_BigInt *gcd,
   [out]   TEE_BigInt *u,
   [out]   TEE_BigInt *v,
   [in]    TEE_BigInt *op1,
   [in]    TEE_BigInt *op2
)
```

Description

The `TEE_BigIntComputeExtendedGcd` function computes the greatest common divisor of the input parameters `op1` and `op2`. Furthermore it computes the coefficients u and v such that `u*op1+v*op2==gcd`. It is allowed that `op1` and `op2` point to the same memory region. It is allowed that u, v, or both can be `NULL`. If both of them are `NULL` then the function only computes the gcd of `op1` and `op2`.

Parameters

- `gcd`: Pointer to `TEE_BigInt` to hold the greatest common divisor of `op1` and `op2`
- u: Pointer to `TEE_BigInt` to hold the first coefficient
- v: Pointer to `TEE_BigInt` to hold the second coefficient
- `op1`: Pointer to the first operand
- `op2`: Pointer to the second operand

Result Sizes

- The `gcd` result must be able to hold `max(magnitude(op1), magnitude(op2))` bits.[7]
- The absolute value of u is in the range `[0, |op2/gcd|-1]`.[8]
- The absolute value of v is in the range `[0, |op1/gcd|-1]`.

[7] The magnitude function is defined in section 8.7.5.

[8] The notation |x| means the absolute value of x.

8.10.3 TEE_BigIntIsProbablePrime

```
int32_t TEE_BigIntIsProbablePrime(
     [in] TEE_BigInt *op,
          uint32_t confidenceLevel
)
```

Description

The `TEE_BigIntIsProbablePrime` function performs a probabilistic primality test on `op`. The parameter `confidenceLevel` is used to specify the probability of a non-conclusive answer. If the function cannot guarantee that `op` is prime or composite, it MUST iterate the test until the probability that `op` is composite is less than `2^(-confidenceLevel)`. Values smaller than 80 for `confidenceLevel` will not be recognized and will default to 80. The maximum honored value of `confidenceLevel` is implementation specific, but MUST be at least 80.

The algorithm for performing the primality test is implementation specific, but its correctness and efficiency MUST be equal to or better than the Miller-Rabin test.

Parameters

- `op`: Candidate number that is tested for primality
- `confidenceLevel`: The desired confidence level for a non-conclusive test. This parameter (usually) maps to the number of iterations and thus to the running time of the test. Values smaller than 80 will be treated as 80.

Return Value

- 0: If op is a composite number
- 1: If `op` is guaranteed to be prime
- -1: If the test is non-conclusive but the probability that `op` is composite is less than `2^(-confidenceLevel)`

8.11 Fast Modular Multiplication Operations

This part of the API allows the implementer of the TEE Internal API to give the TA developer access to faster modular multiplication routines, possible hardware accelerated. These functions MAY be implemented using Montgomery or Barrett or any other suitable technique for fast modular multiplication. If no such support is possible the functions in this section MAY be implemented using regular multiplication and modular reduction. The data type `TEE_BigIntFMM` is used to represent the integers during repeated multiplications such as when calculating a modular exponentiation. The internal representation of the `TEE_BigIntFMM` is implementation specific.

8.11.1 TEE_BigIntConvertToFMM

```
void TEE_BigIntConvertToFMM(
   [out]   TEE_BigIntFMM          *dest,
   [in]    TEE_BigInt             *src,
   [in]    TEE_BigInt             *n,
   [in]    TEE_BigIntFMMContext   *context
)
```

Description

The `TEE_BigIntConvertToFMM` function converts `src` into a representation suitable for doing fast modular multiplication. If the operation is successful, the result will be written in implementation specific format into the buffer `dest`, which must have been allocated by the TA and initialized using `TEE_BigIntInitFMM`.

Parameters

- `dest`: Pointer to an initialized `TEE_BigIntFMM` memory area
- `src`: Pointer to the `TEE_BigInt` to convert
- `n`: Pointer to the modulus
- `context`: Pointer to a context previously initialized using `TEE_BigIntInitFMMContext`

8.11.2 TEE_BigIntConvertFromFMM

```
void TEE_BigIntConvertFromFMM(
   [out]   TEE_BigInt              *dest,
   [in]    TEE_BigIntFMM           *src,
   [in]    TEE_BigInt              *n,
   [in]    TEE_BigIntFMMContext    *context,
)
```

Description

The `TEE_BigIntConvertFromFMM` function converts `src` in the fast modular multiplication representation back to a `TEE_BigInt` representation.

Parameters

- `dest`: Pointer to an initialized `TEE_BigInt` memory area to hold the converted result
- `src`: Pointer to a `TEE_BigIntFMM` holding the value in the fast modular multiplication representation
- `n`: Pointer to the modulus
- `context`: Pointer to a context previously initialized using `TEE_BigIntInitFMMContext`

8.11.3 TEE_BigIntComputeFMM

```
void TEE_BigIntComputeFMM(
   [out]   TEE_BigIntFMM           *dest,
   [in]    TEE_BigIntFMM           *op1,
   [in]    TEE_BigIntFMM           *op2,
   [in]    TEE_BigInt              *n,
   [in]    TEE_BigIntFMMContext    *context,
)
```

Description

The `TEE_BigIntComputeFMM` function calculates `dest` = `op1` * `op2` in the fast modular multiplication representation. The pointers `dest`, `op1` and `op2` MUST all point to a `TEE_BigIntFMM` which has been previously initialized with the same modulus and context as used in this function call; otherwise the result is undefined. It is allowed that all or some of `dest`, `op1`, and `op2` point to the same memory region.

Parameters

- `dest`: Pointer to `TEE_BigIntFMM` to hold the result `op1` * `op2` in the fast modular multiplication representation
- `op1`: Pointer to the first operand
- `op2`: Pointer to the second operand
- `n`: Pointer to the modulus
- `context`: Pointer to a context previously initialized using `TEE_BigIntInitFMMContext`

Functions

Functions by Category