Formal Specification of Trusted Execution Environment APIs

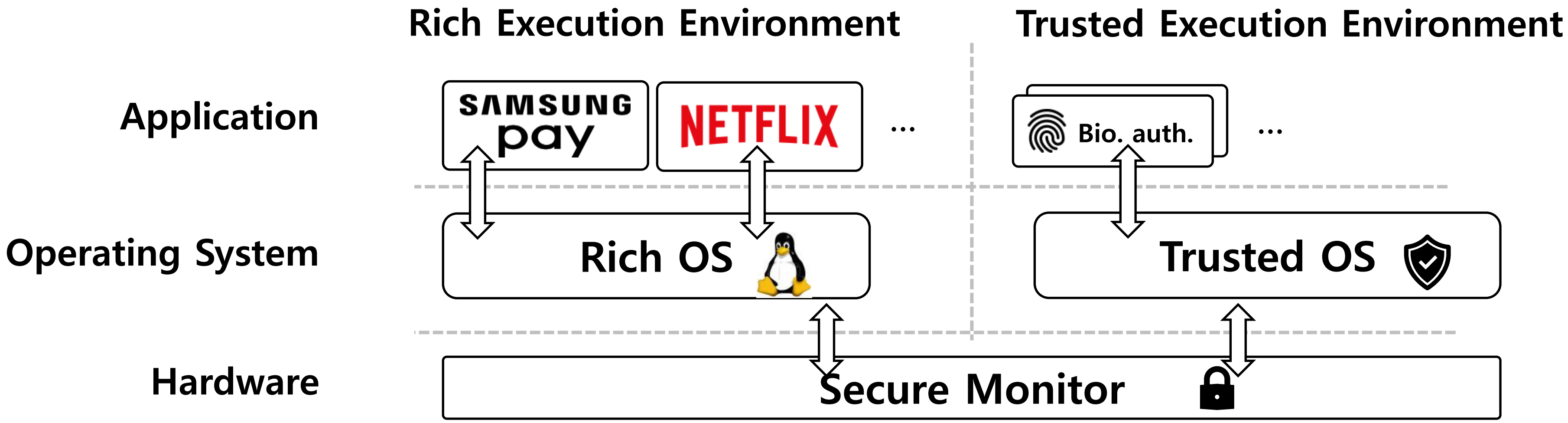
Geunyeol Yu[1] Seunghyun Chae[1] Kyungmin Bae[1] Sungkun Moon[2]

FASE2024

[1] POSTECH [2] SAMSUNG

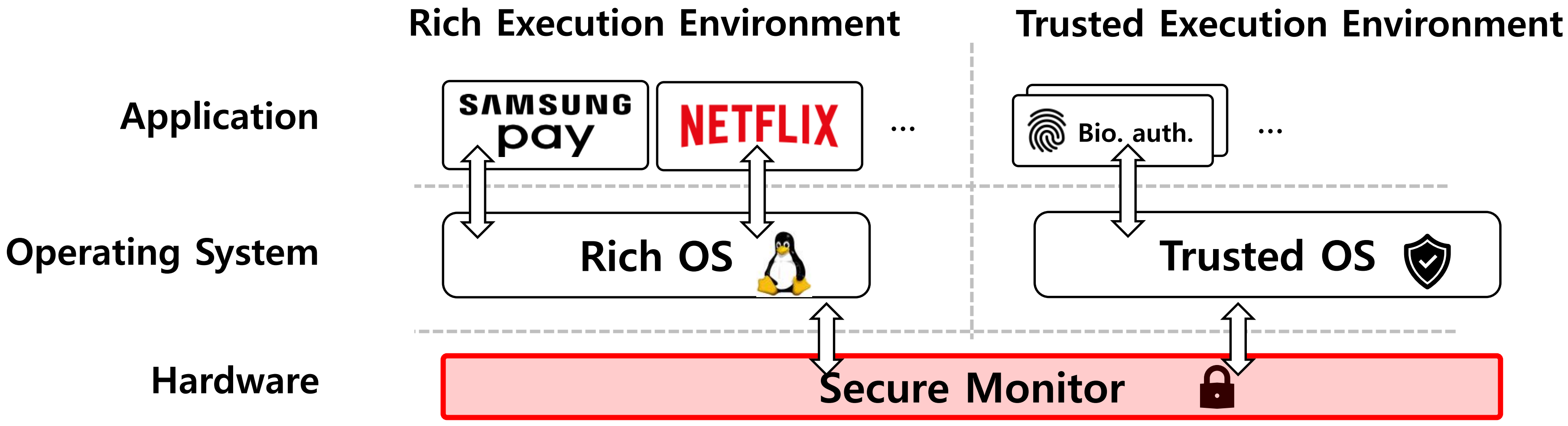
Trusted Execution Environment

- Trusted execution environment (TEE) is a physically isolated execution environment for securing sensitive computations.



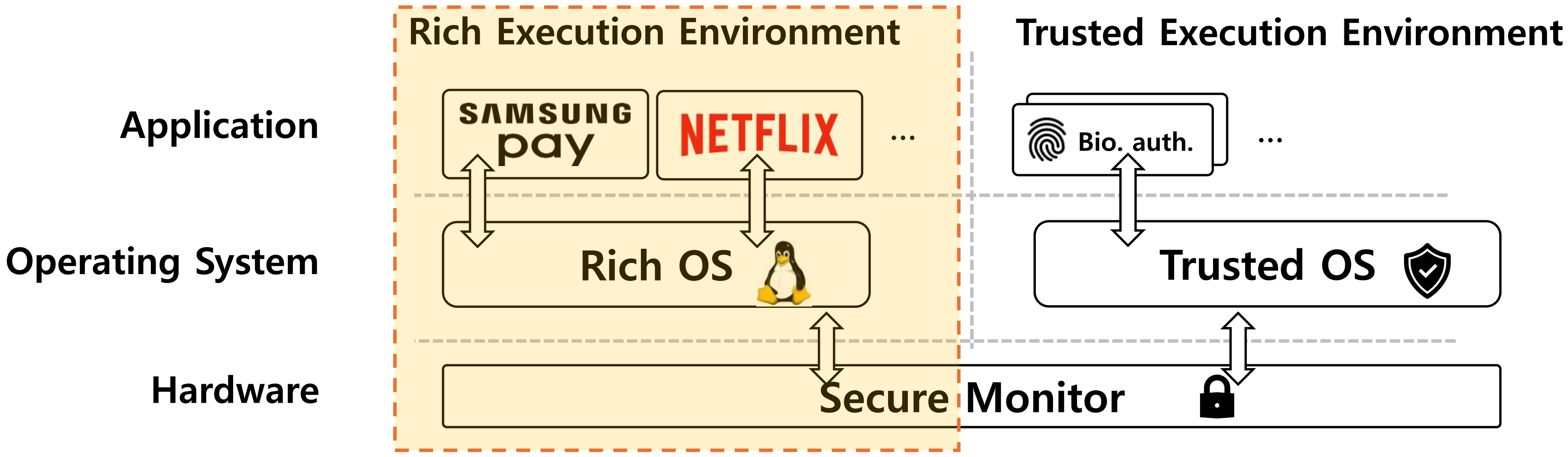
Trusted Execution Environment

- Trusted execution environment (TEE) is a physically isolated execution environment for securing sensitive computations.



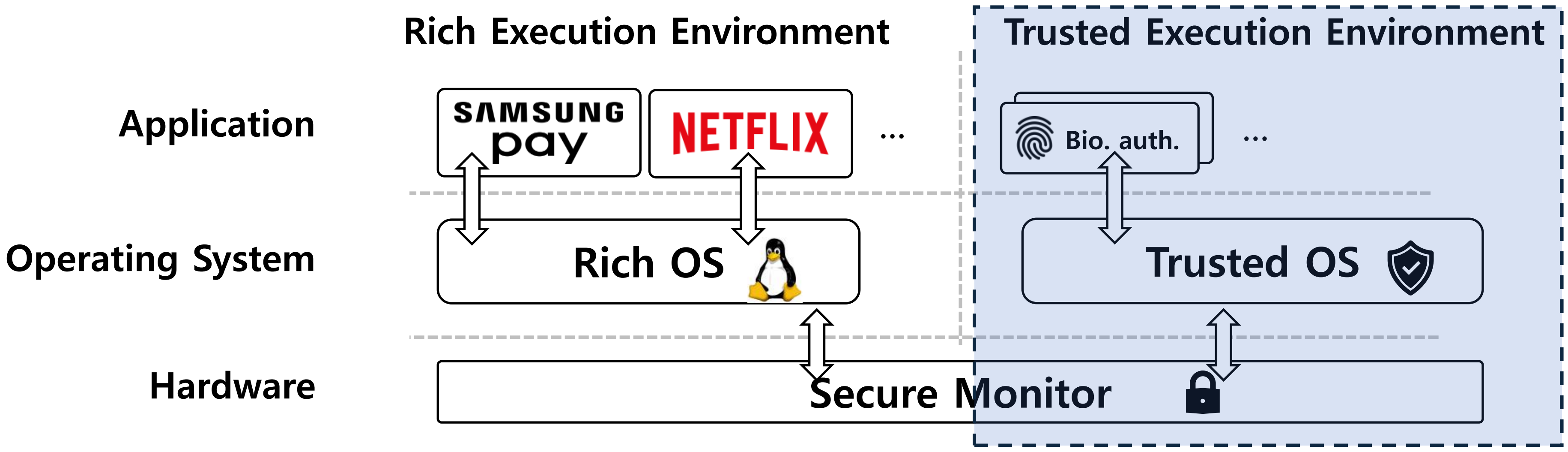
Trusted Execution Environment

- Trusted execution environment (TEE) is a physically isolated execution environment for securing sensitive computations.



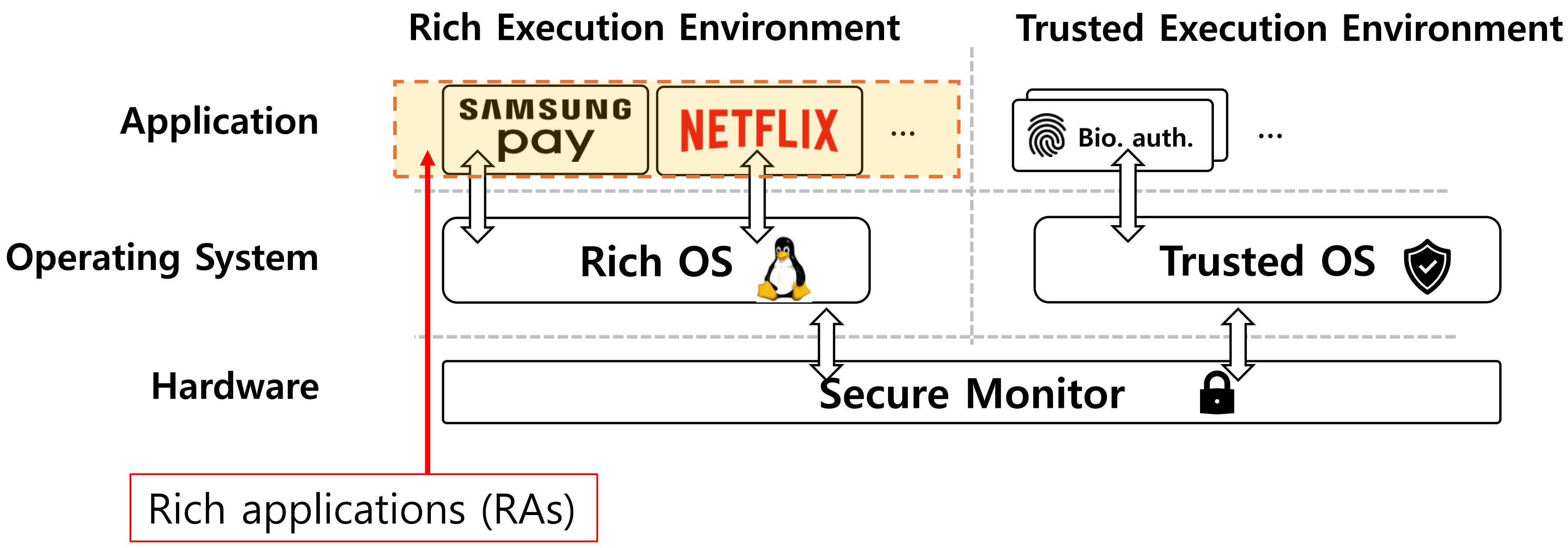
Trusted Execution Environment

- Trusted execution environment (TEE) is a physically isolated execution environment for securing sensitive computations.



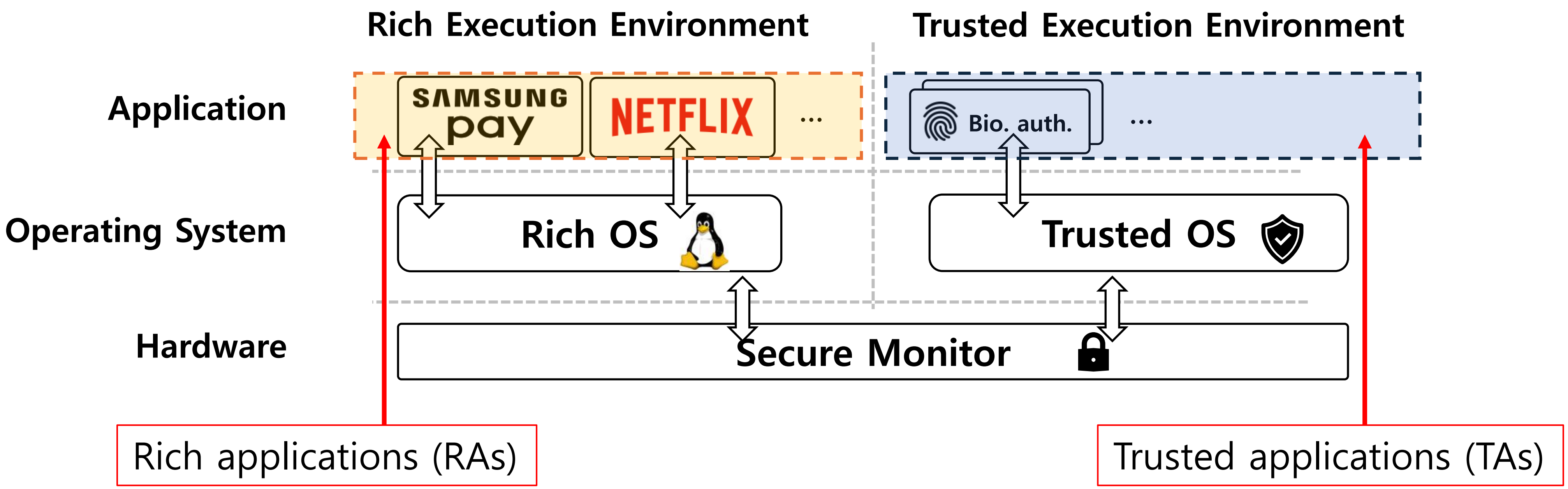
Trusted Execution Environment

- Trusted execution environment (TEE) is a physically isolated execution environment for securing sensitive computations.



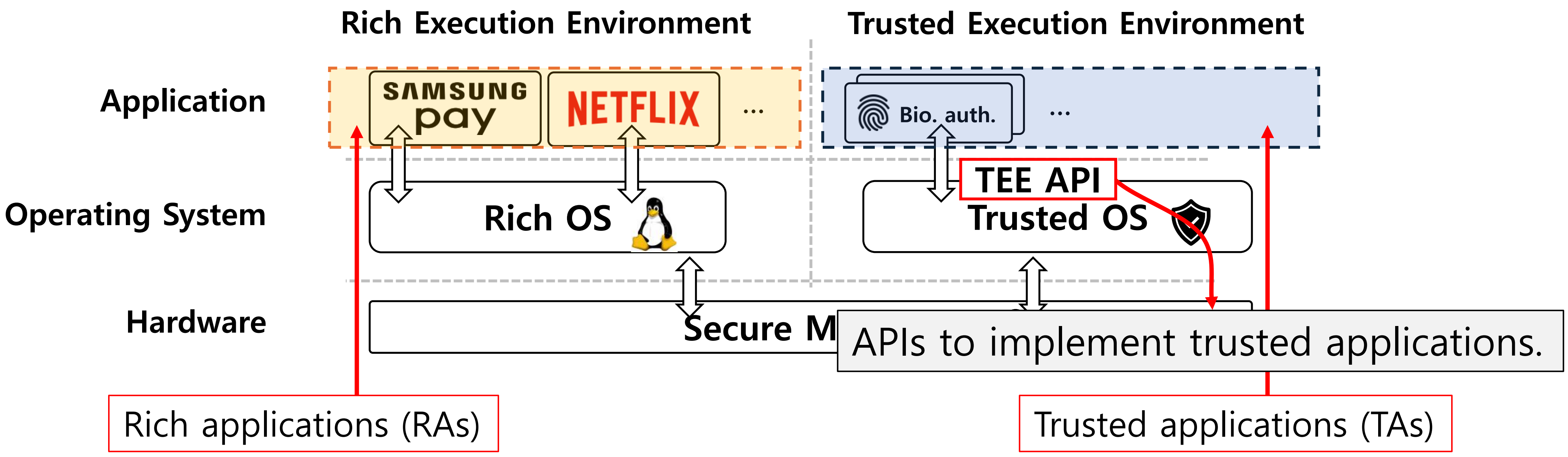
Trusted Execution Environment

- Trusted execution environment (TEE) is a physically isolated execution environment for <u>securing sensitive computations</u>.



Trusted Execution Environment

- Trusted execution environment (TEE) is a physically isolated execution environment for securing sensitive computations.



Trusted Execution Environment

- Because TEE is physically isolated environment, it guarantees the integrity and confidentiality of executed programs and their data.

Trusted Execution Environment

- Because TEE is physically isolated environment, it guarantees the integrity and confidentiality of executed programs and their data.

- This is why TEE is widely used in security-critical systems, such as industrial control systems, servers, mobile security, IoT, etc.

Motivations

- Formal analysis framework for TEE applications is not well-developed.

Motivations

- Formal analysis framework for TEE applications is not well-developed.

- Formal models for TEE and its APIs, which can be utilized for a variety of formal analysis techniques, are lacking.

Our Contributions

- We provide a comprehensive formal model for TEE APIs, that can be used in various formal analysis.

- We specify two widely used TEE API categories, Trusted Storage API and Cryptographic Operations API.

- We demonstrate the effectiveness of our model through a case study on formally analyzing a real-world TEE application, MQT-TZ.
  - Identify security vulnerabilities in the MQT-TZ implementation.
  - Patch them and verify the fix with model checking.

Our Contributions

- We provide a comprehensive formal model for TEE APIs, that can be used in various formal analysis.

- We specify two widely used TEE API categories, Trusted Storage API and Cryptographic Operations API.

- We demonstrate the effectiveness of our model through a case study on formally analyzing a real-world TEE application, MQT-TZ.
  - Identify security vulnerabilities in the MQT-TZ implementation.
  - Patch them and verify the fix with model checking.

Our Contributions

- We provide a comprehensive formal model for TEE APIs, that can be used in various formal analysis.
- We specify two widely used TEE API categories, Trusted Storage API and Cryptographic Operations API.
- We demonstrate the effectiveness of our model through a case study on formally analyzing a real-world TEE application, MQT-TZ.

  - Identify security vulnerabilities in the MQT-TZ implementation.
  - Patch them and verify the fix with model checking.

Our Contributions

- We provide a comprehensive formal model for TEE APIs, that can be used in various formal analysis.

- We specify two widely used TEE API categories, Trusted Storage API and Cryptographic Operations API.

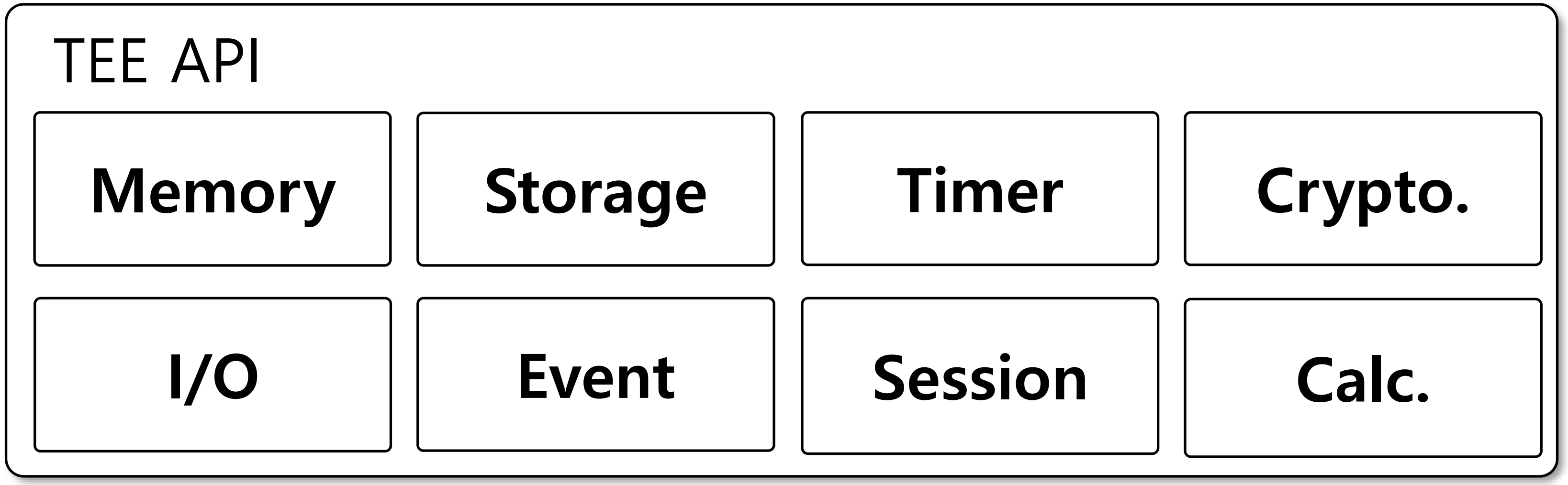
- We demonstrate the effectiveness of our model through a case study on formally analyzing a real-world TEE application, MQT-TZ.
  - Identify security vulnerabilities in the MQT-TZ implementation.
  - Patch them and verify the fix with model checking.

Our Contributions

- We provide a comprehensive formal model for TEE APIs, that can be used in various formal analysis.
- We specify two widely used TEE API categories, Trusted Storage API and Cryptographic Operations API.
- We demonstrate the effectiveness of our model through a case study on formally analyzing a real-world TEE application, MQT-TZ.
  - Identify security vulnerabilities in the MQT-TZ implementation.
  - Patch them and verify the fix with model checking.

Our Target TEE APIs

- Our target is the standard TEE APIs, provided by Global Platform.
  - Many Trusted OSes follow this standard.
  - e.g., Samsung TEEgris, Trustonic Kinibi, Qualcomm QTEE, etc.



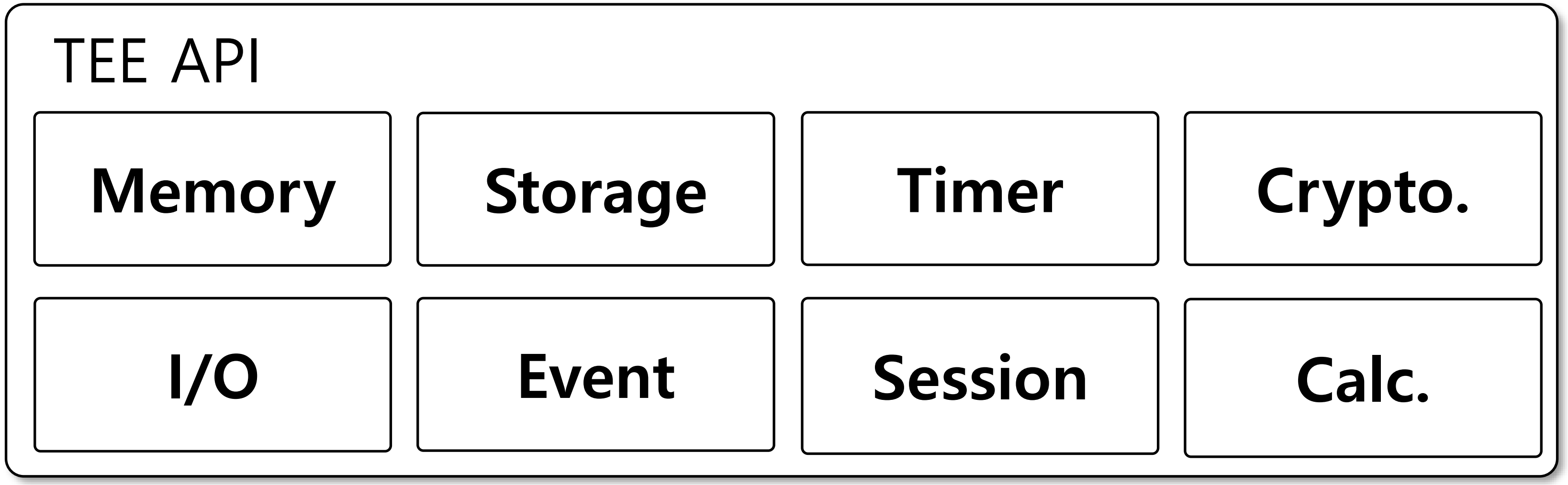

GLOBALPLATFORM
THE STANDARD FOR SECURE DIGITAL SERVICES AND DEVICES

GlobalPlatform Technology
TEE Internal Core API Specification
Version 1.3.1

Public Release
July 2021
Document Reference: GPD_SPE_010

Our Target TEE APIs

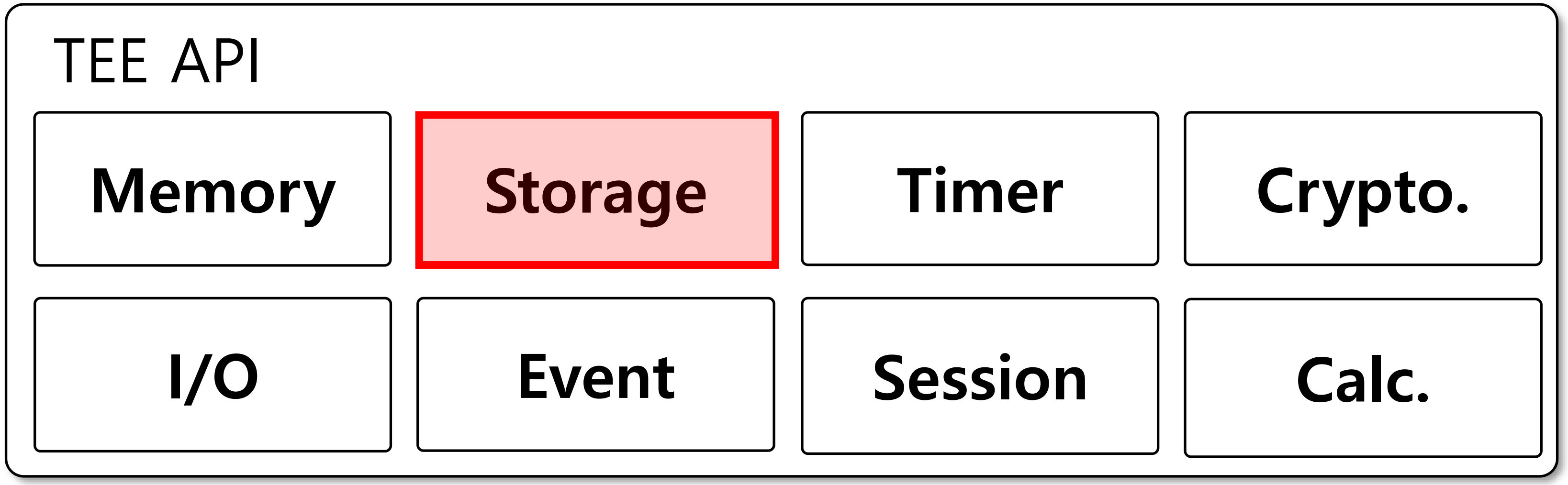
- We focus on Trusted Storage API and Cryptographic Operations API.



Our Target TEE APIs

- We focus on Trusted Storage API and Cryptographic Operations API.

Manges files and crypto keys in trusted storage




GLOBALPLATFORM

GlobalPlatform Technology
TEE Internal Core API Specification
Version 1.3.1

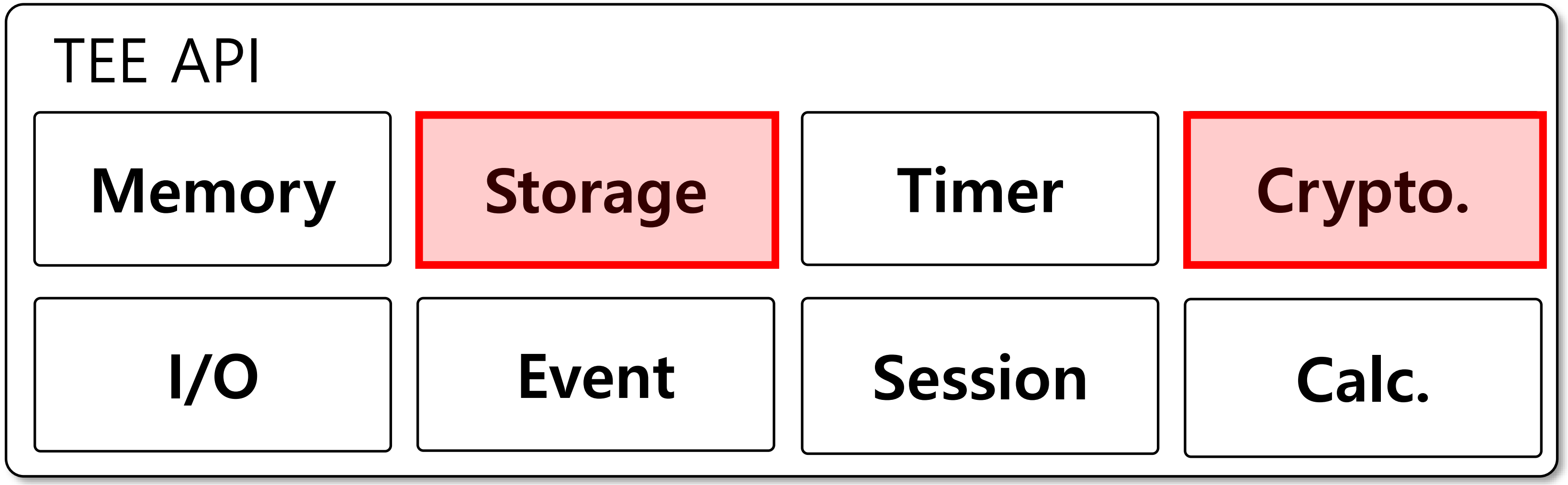
Public Release
July 2021
Document Reference: GPD_SPE_010

Our Target TEE APIs

- We focus on Trusted Storage API and Cryptographic Operations API.

Manges files and crypto keys in trusted storage

Handles cryptographic algorithms



Our Target TEE APIs

- We focus on Trusted Storage API and Cryptographic Operations API.

Manges files and keys in trusted storage

Handles cryptographic algorithms

- We choose these APIs because:
  - They are widely and frequently used in various TEE applications;
  - They provide essential functions for TEE's integrity.

I/O

Event

Session

Calc.

Characteristics of the TEE APIs

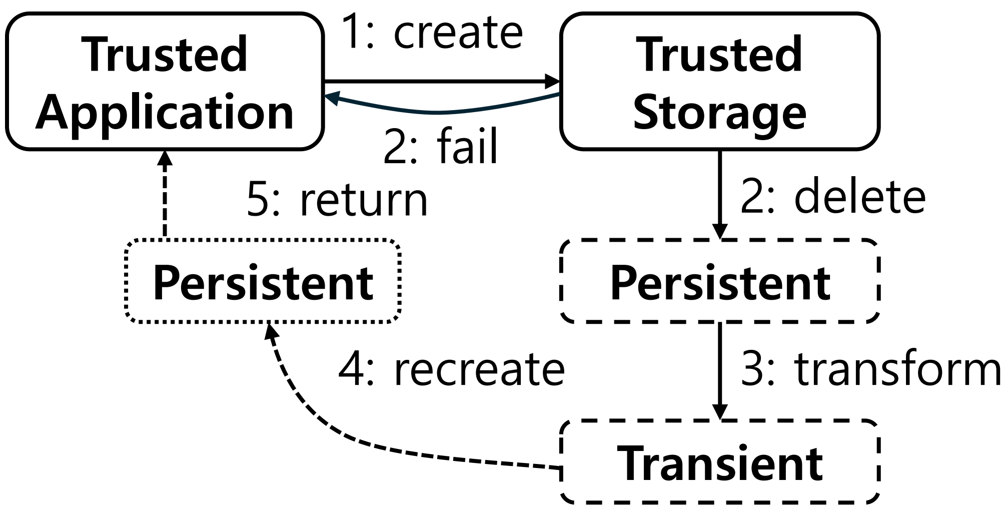
Characteristics of the TEE APIs

- (1) Many API functions interact with multiple objects, and we need to consider their concurrent behaviors.

Characteristics of the TEE APIs

- (1) Many API functions interact with multiple objects, and we need to consider their <u>concurrent behaviors</u>.

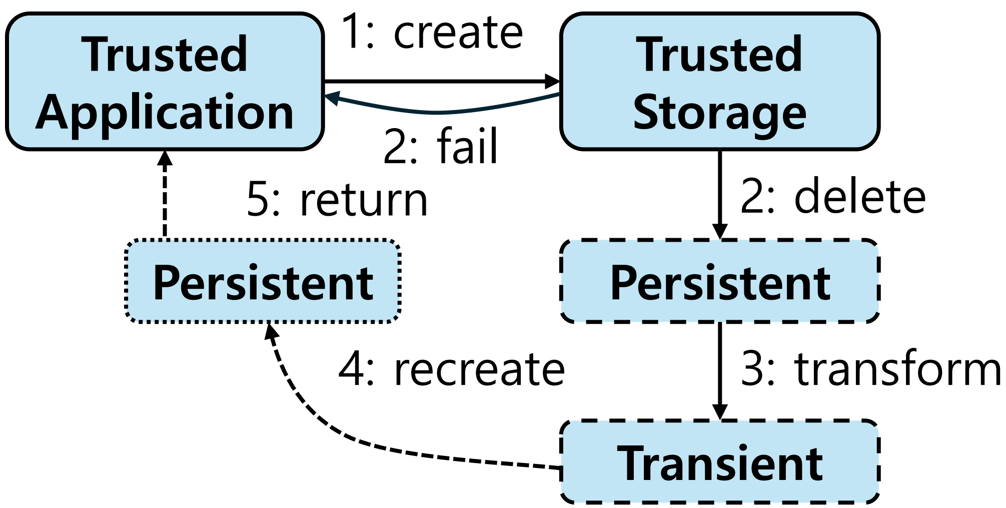
- E.g., consider a file open function of Trusted Storage API.



Characteristics of the TEE APIs

- (1) Many API functions interact with multiple objects, and we need to consider their concurrent behaviors.

- E.g., consider a file open function of Trusted Storage API.



Characteristics of the TEE APIs

- (2) Some objects have complex internal state transitions.

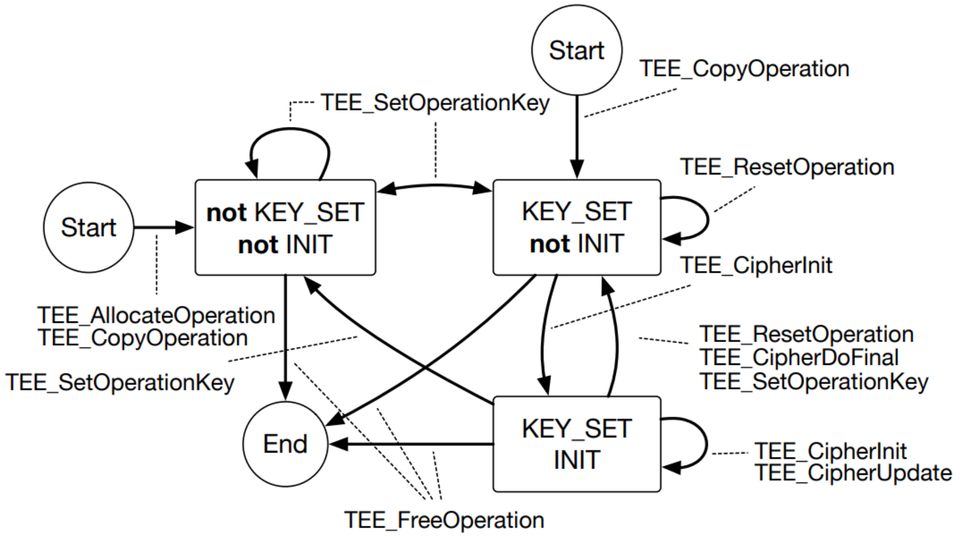
Characteristics of the TEE APIs

- (2) Some objects have complex internal state transitions.

- E.g., A symmetric cipher operation object has complex state transitions.

Characteristics of the TEE APIs

- (2) Some objects have complex internal state transitions.
- E.g., A symmetric cipher operation object has complex state transitions.



Characteristics of the TEE APIs

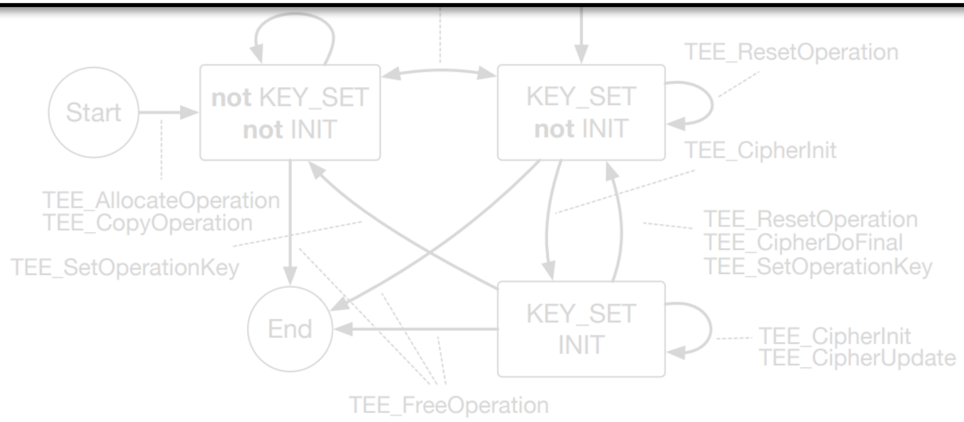
- (2) Some objects have complex internal state transitions.
- E.g., A symmetric cipher operation object has complex state transitions.

Characteristics of the TEE APIs

- (2) Some objects have complex internal state transitions.

- E.g., A symmetric cipher operation object has complex state transitions.

Considering these characteristics, we use Maude for formal specification.



What is Maude?

- Maude is a language and tool for formally specifying and analyzing concurrent systems, based on rewriting logic formalism.

What is Maude?

- Maude is a language and tool for formally specifying and analyzing concurrent systems, based on rewriting logic formalism.
  - It supports object-oriented specification.

What is Maude?

- Maude is a language and tool for formally specifying and analyzing concurrent systems, based on rewriting logic formalism.
  - It supports object-oriented specification.
  - It defines concurrent behaviors using rewrite rules.

What is Maude?

- Maude is a language and tool for formally specifying and analyzing concurrent systems, based on rewriting logic formalism.
  - It supports object-oriented specification.
  - It defines concurrent behaviors using rewrite rules.

We can formally specify TEE APIs considering characteristic 1 and 2.

What is Maude?

- Maude is a language and tool for formally specifying and analyzing concurrent systems, based on rewriting logic formalism.
  - It supports object-oriented specification.
  - It defines concurrent behaviors using rewrite rules.

- Because of the powerful formalism of Maude, it is widely used in various formal analysis domains such as:
  - defining language semantics,
  - inductive theorem proving,
  - model checking, etc.

Formal Specification using Maude

Formal Specification using Maude

- In Maude, we declare a class using the syntax:

Class name

$$\textbf{class}\ \texttt{C}\ |\ \texttt{att}_1 : \texttt{Ty}_1\ ,\ \texttt{...,}\ \texttt{att}_n : \texttt{Ty}_n$$

Attributes and their types

Formal Specification using Maude

- In Maude, we declare class instances using the syntax:

$$\texttt{\textbf{class} C | att}_1 \texttt{ : Ty}_1 \texttt{ , ..., att}_n \texttt{ : Ty}_n$$

- The behavior of a class is defined using rewrite rules:

$$\textbf{crl}\ [label] : l \Rightarrow r\ \textbf{if}\ \phi$$

Formal Specification using Maude

- In Maude, we declare class instances using the syntax:

```
class C | att₁ : Ty₁ , ..., attₙ : Tyₙ
```

- The behavior of a class is defined using rewrite rules:

$$\textbf{crl}\ [label] : l \Rightarrow r\ \textbf{if}\ \phi$$

Pattern

Formal Specification using Maude

- In Maude, we declare class instances using the syntax:

$$\texttt{class C | att}_1 \texttt{ : Ty}_1 \texttt{ , ..., att}_n \texttt{ : Ty}_n$$

- The behavior of a class is defined using rewrite rules:

$$\mathbf{crl}\ [label] : l \Rightarrow r\ \mathbf{if}\ \phi$$

Pattern

Rewrites to

Formal Specification using Maude

- In Maude, we declare class instances using the syntax:

$$\texttt{class C | att}_1 \texttt{ : Ty}_1 \texttt{ , ..., att}_n \texttt{ : Ty}_n$$

- The behavior of a class is defined using rewrite rules:

condition

$$\textbf{crl}\ [label] : l \Rightarrow r\ \textbf{if}\ \phi$$

Pattern Rewrites to

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

```
rl [read]:
   (msg reqRead from TA to PI)
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

```
rl [read]:
   (msg reqRead from TA to PI)        Message object
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

```
rl [read]:
   (msg reqRead from TA to PI)
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Message object

persistent object

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

```
rl [read]:
   (msg reqRead from TA to PI)
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Message object

persistent object

persistent object

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

```
rl [read]:
   (msg reqRead from TA to PI)
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Message object

persistent object

persistent object

Message object

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

Persistent object →

```
rl [read]:
   (msg reqRead from TA to PI)
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

```
rl [read]:
   (msg reqRead from TA to PI)
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Read request message

Persistent object

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

```
rl [read]:
   (msg reqRead from TA to PI)
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Read request message

Persistent object

Pop a top element

Formal Specification using Maude

- E.g.) In TEE, a file is called a persistent object having:
  - (1) a file name; and
  - (2) a data stream.

```
class PersistObj | file-name : String, data-stream : List{Data}
```

- This object returns its data when receiving a read request message.

```
rl [read]:
   (msg reqRead from TA to PI)
   < PI : PersistObj | file-name : FILE, data-stream : DATA :: STREAM >
=> < PI : PersistObj | file-name : FILE, data-stream : STREAM >
   (msg retData[DATA] from PI to TK)
```

Read request message

Persistent object

Pop a top element

Returns the element

An example: `TEE_CreatePersistentObject`

An example: `TEE_CreatePersistentObject`

- This function creates a new persistent object.

File

An example: `TEE_CreatePersistentObject`

- This function creates a new persistent object.

  - Argument 1 : Filename
  - Argument 2 : Access flags (e.g., overwrite)
  - Argument 3 : Data
  - …

An example: `TEE_CreatePersistentObject`

- This function creates a new persistent object.

  - Argument 1 : Filename
  - Argument 2 : Access flags (e.g., overwrite)
  - Argument 3 : Data
  - ...

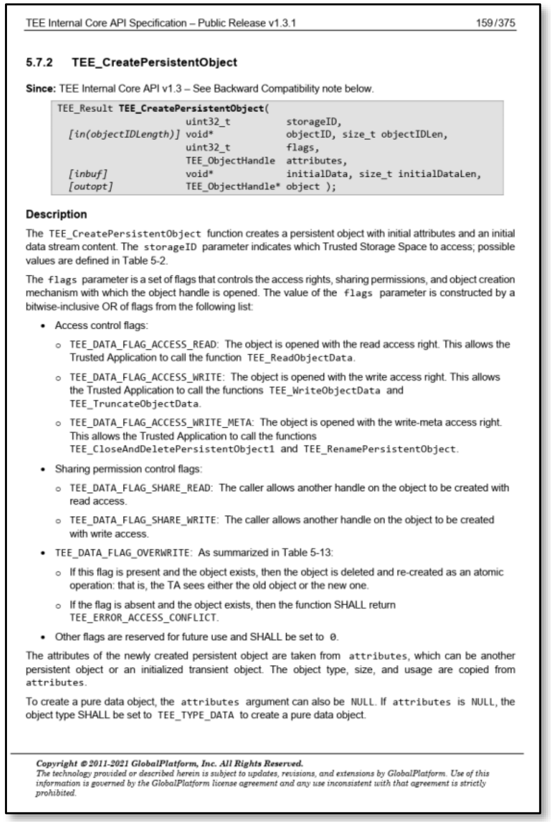
It's a file open function but opens the file to a trusted storage.

An example: `TEE_CreatePersistentObject`

- According to the TEE API document, when a file with the same name already exists, the behavior of the function is as follows:

An example: `TEE_CreatePersistentObject`

- According to the TEE API document, when a file with the same name already exists, the behavior of the function is as follows:



5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]             TEE_ObjectHandle* object );
```

Description

The `TEE_CreatePersistentObject` function creates a persistent object with initial attributes and an initial data stream content. The `storageID` parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

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_CloseAndDeletePersistentObject1` 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_OVERWRITE`: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return `TEE_ERROR_ACCESS_CONFLICT`.
- Other flags are reserved for future use and SHALL be set to `0`.

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

To create a pure data object, the `attributes` argument can also be `NULL`. If `attributes` is `NULL`, the object type SHALL be set to `TEE_TYPE_DATA` to create a pure data object.

An example: `TEE_CreatePersistentObject`

- According to the TEE API document, when a file with the same name already exists, the behavior of the function is as follows:

TEE Internal Core API Specification – Public Release v1.3.1 159/375

5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]                TEE_ObjectHandle* object );
```

Description

The `TEE_CreatePersistentObject` function creates a persistent object with initial attributes and an initial data stream content. The `storageID` parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

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_CloseAndDeletePersistentObject1` 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_OVERWRITE`: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return `TEE_ERROR_ACCESS_CONFLICT`.
- Other flags are reserved for future use and SHALL be set to `0`.

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

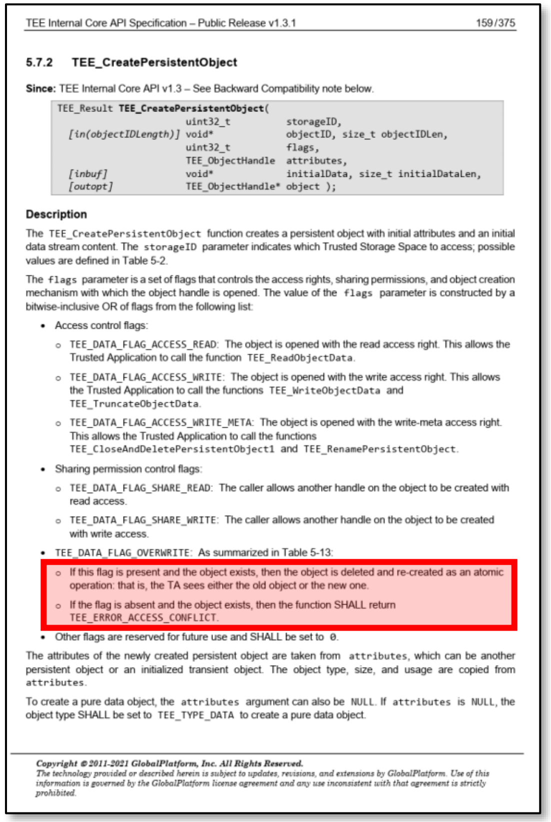
To create a pure data object, the `attributes` argument can also be `NULL`. If `attributes` is `NULL`, the object type SHALL be set to `TEE_TYPE_DATA` to create a pure data object.

An example: TEE_CreatePersistentObject

- According to the TEE API document, when a file with the same name already exists, the behavior of the function is as follows:

  - Overwrite flag given :

    Delete the old file and create a new one



5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]              TEE_ObjectHandle*  object );
```

Description

The TEE_CreatePersistentObject function creates a persistent object with initial attributes and an initial data stream content. The storageID parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

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_CloseAndDeletePersistentObject1 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_OVERWRITE: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return TEE_ERROR_ACCESS_CONFLICT.
- Other flags are reserved for future use and SHALL be set to 0.

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

To create a pure data object, the attributes argument can also be NULL. If attributes is NULL, the object type SHALL be set to TEE_TYPE_DATA to create a pure data object.

An example: TEE_CreatePersistentObject

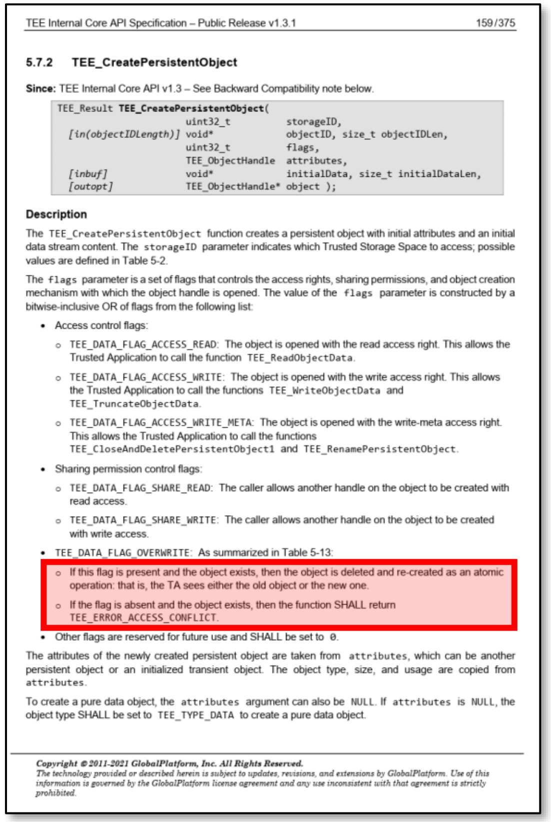
- According to the TEE API document, when a file with the same name already exists, the behavior of the function is as follows:

  - Overwrite flag given :

    Delete the old file and create a new one

  - Overwrite flag not given :

    Return error



5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]            TEE_ObjectHandle*  object );
```

Description

The TEE_CreatePersistentObject function creates a persistent object with initial attributes and an initial data stream content. The storageID parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

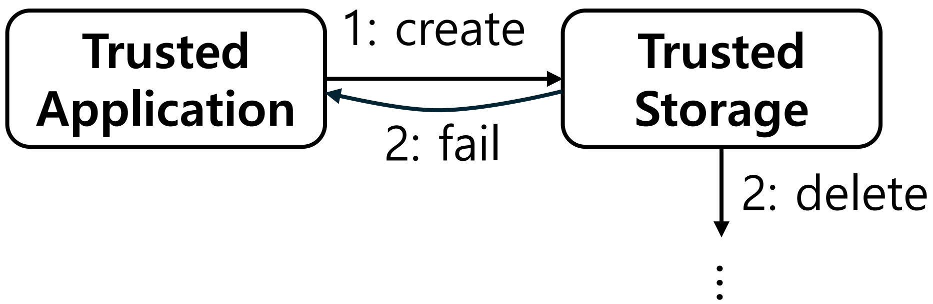
The flags parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the flags parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- 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_CloseAndDeletePersistentObject1 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_OVERWRITE: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return TEE_ERROR_ACCESS_CONFLICT.
- Other flags are reserved for future use and SHALL be set to 0.

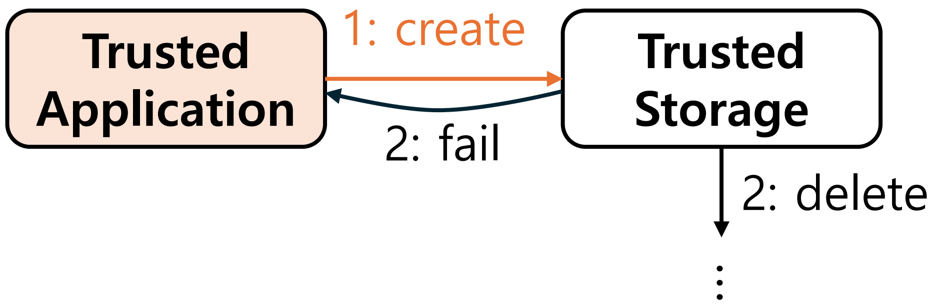
The attributes of the newly created persistent object are taken from attributes, which can be another persistent object or an initialized transient object. The object type, size, and usage are copied from attributes.

To create a pure data object, the attributes argument can also be NULL. If attributes is NULL, the object type SHALL be set to TEE_TYPE_DATA to create a pure data object.

An example: `TEE_CreatePersistentObject`

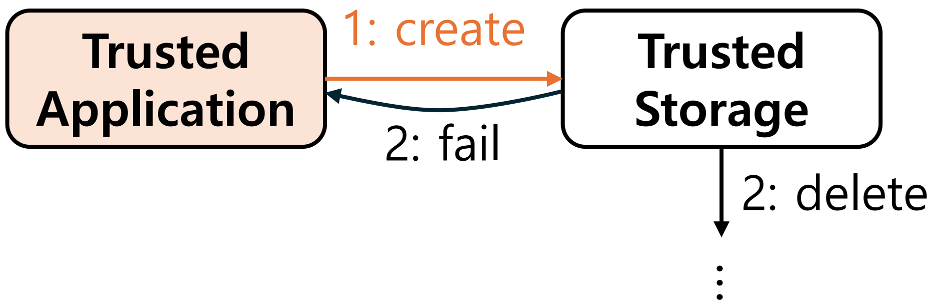


An example: `TEE_CreatePersistentObject`

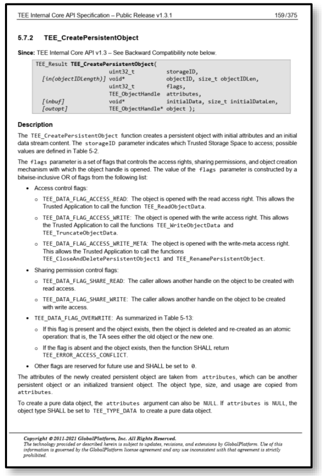


- (1) A trusted application (TA) requests a trusted storage to create a file.

An example: TEE_CreatePersistentObject



- (1) A trusted application (TA) requests a trusted storage to create a file.



5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]            TEE_ObjectHandle*   object );
```

Description

The TEE_CreatePersistentObject function creates a persistent object with initial attributes and an initial data stream content. The storageID parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

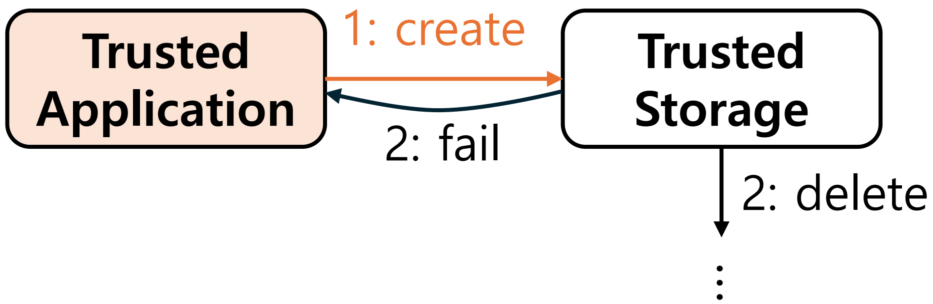
The flags parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the flags parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- 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_CloseAndDeletePersistentObject1 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_OVERWRITE: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return TEE_ERROR_ACCESS_CONFLICT.
- Other flags are reserved for future use and SHALL be set to 0.

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

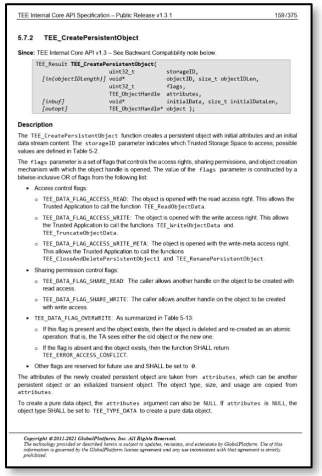
To create a pure data object, the attributes argument can also be NULL. If attributes is NULL, the object type SHALL be set to TEE_TYPE_DATA to create a pure data object.

An example: `TEE_CreatePersistentObject`



- (1) A trusted application (TA) requests a trusted storage to create a file.
- Trusted application has the following things:

  - the status of an API call,
  - an identifier of a trusted storage,
  - …


TEE Internal Core API Specification – Public Release v1.3.1 159/375

5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]                TEE_ObjectHandle* object );
```

Description

The TEE_CreatePersistentObject function creates a persistent object with initial attributes and an initial data stream content. The storageID parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

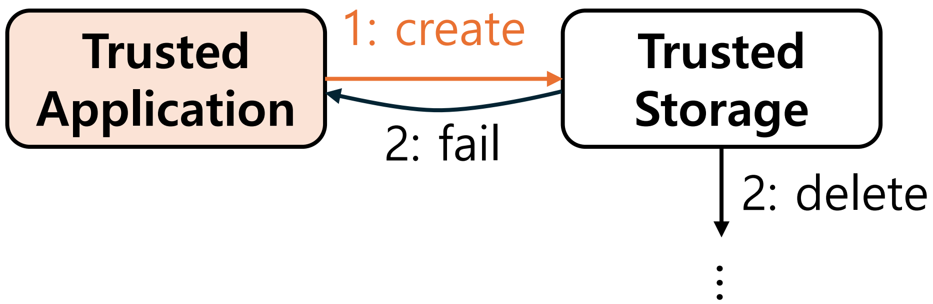
The flags parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the flags parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- 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_CloseAndDeletePersistentObject1 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_OVERWRITE: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return TEE_ERROR_ACCESS_CONFLICT.
- Other flags are reserved for future use and SHALL be set to 0.

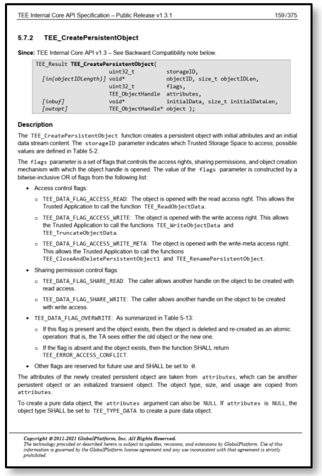
The attributes of the newly created persistent object are taken from attributes, which can be another persistent object or an initialized transient object. The object type, size, and usage are copied from attributes.

To create a pure data object, the attributes argument can also be NULL. If attributes is NULL, the object type SHALL be set to TEE_TYPE_DATA to create a pure data object.

An example: `TEE_CreatePersistentObject`



- (1) A trusted application (TA) requests a trusted storage to create a file.
- Trusted application has the following things:
  - the status of an API call,
  - an identifier of a trusted storage,
  - …



TEE Internal Core API Specification – Public Release v1.3.1 159/375

5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]           TEE_ObjectHandle*  object );
```

Description

The `TEE_CreatePersistentObject` function creates a persistent object with initial attributes and an initial data stream content. The `storageID` parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

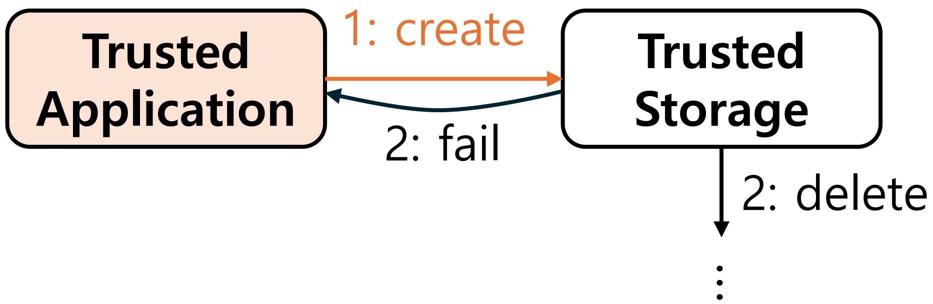
The `flags` parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the `flags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- 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_CloseAndDeletePersistentObject1` 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_OVERWRITE`: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return `TEE_ERROR_ACCESS_CONFLICT`.
- Other flags are reserved for future use and SHALL be set to `0`.

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

To create a pure data object, the `attributes` argument can also be `NULL`. If `attributes` is `NULL`, the object type SHALL be set to `TEE_TYPE_DATA` to create a pure data object.

An example: `TEE_CreatePersistentObject`

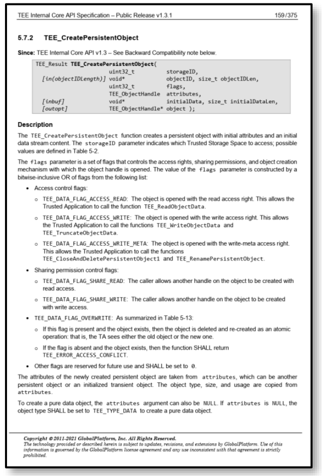


- (1) A trusted application (TA) requests a trusted storage to create a file.

- Trusted application has the following things:

```
class TA | api-call : CallStatus, storage-id : Oid, ...
```

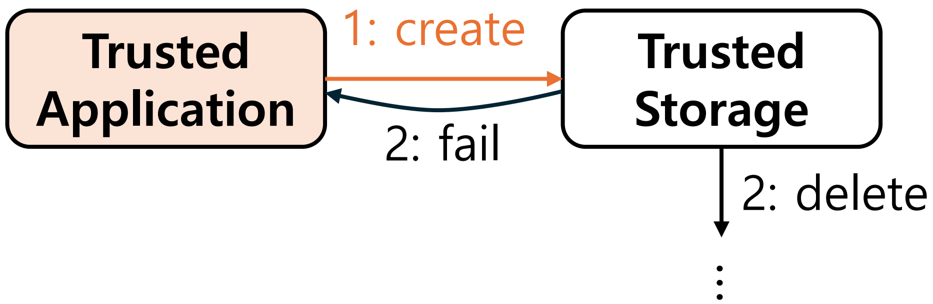
  - the status of an API call,
  - an identifier of a trusted storage,
  - ...



TEE Internal Core API Specification – Public Release v1.3.1

5.7.2 TEE_CreatePersistentObject

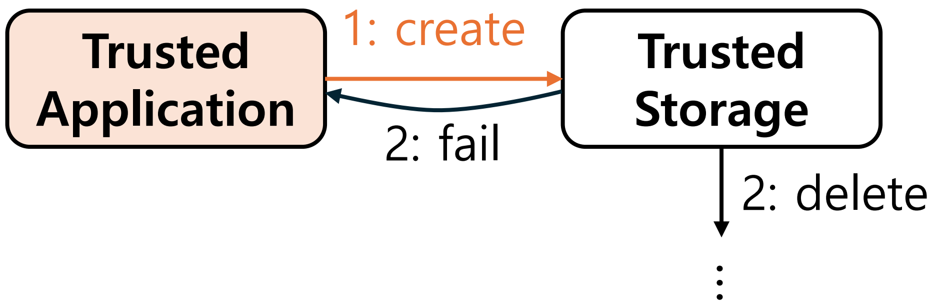
An example: `TEE_CreatePersistentObject`



- (1) A trusted application (TA) requests a trusted storage to create a file.

```
rl [create-persistent-determine-case]:
   < X : TA | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT), storage : SI >
=> < X : TA | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT) # 1 >
   (msg fileCreate[FILE, FLAGS, HI, DATA, OPT] from X to SI) .
```

An example: `TEE_CreatePersistentObject`

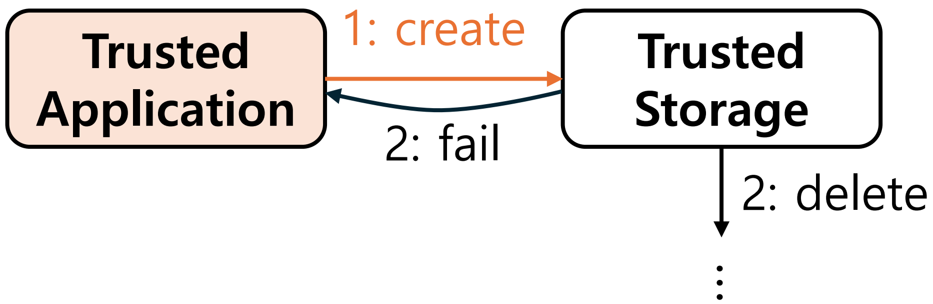


- (1) A trusted application (TA) requests a trusted storage to create a file.

TA

```
rl [create-persistent-determine-case]:
   < X : TA | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT), storage : SI >
=> < X : TA | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT) # 1 >
   (msg fileCreate[FILE, FLAGS, HI, DATA, OPT] from X to SI) .
```

An example: `TEE_CreatePersistentObject`



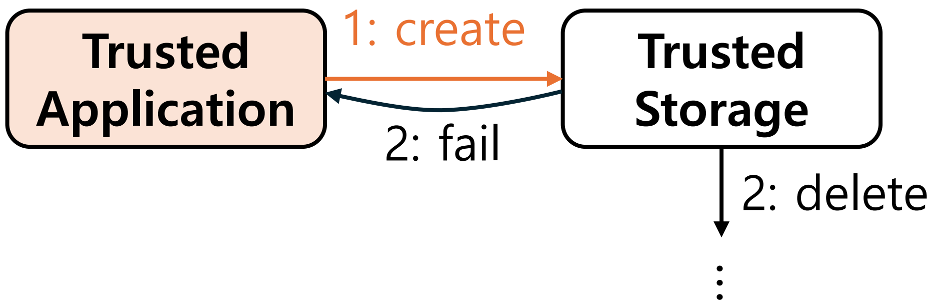
- (1) A trusted application (TA) requests a trusted storage to create a file.

TA

```
rl [create-persistent-determine-case]:
   < X : TA | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT), storage : SI >
=> < X : TA | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT) # 1 >
   (msg fileCreate[FILE, FLAGS, HI, DATA, OPT] from X to SI) .
```

Make a file creation request message and send it

An example: `TEE_CreatePersistentObject`



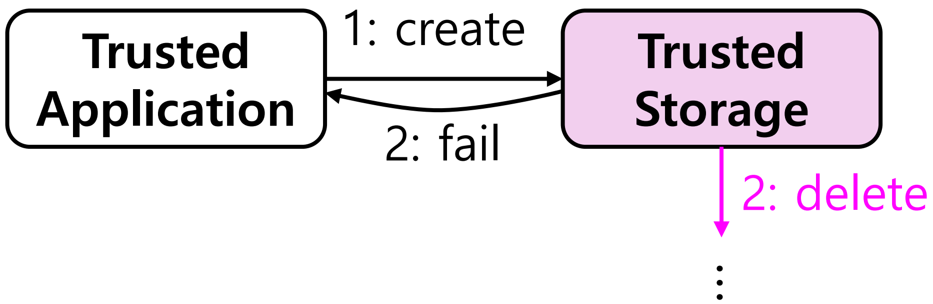
- (1) A trusted application (TA) requests a trusted storage to create a file.

TA

```
rl [create-persistent-determine-case]:
   < X : TA | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT), storage : SI >
=> < X : TA | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT) # 1 >
   (msg fileCreate[FILE, FLAGS, HI, DATA, OPT] from X to SI) .
```

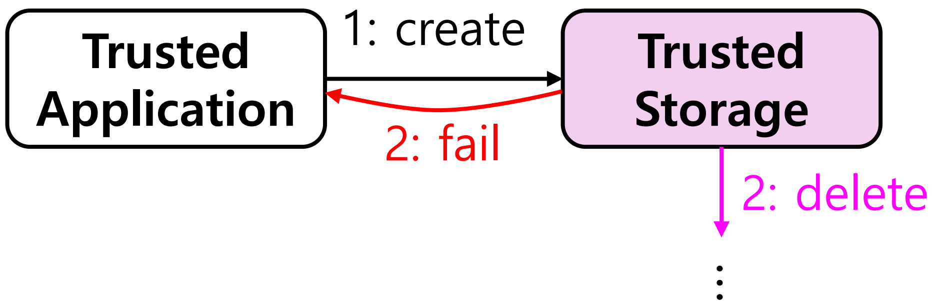
Make a file creation request message and send it to its trusted storage

An example: `TEE_CreatePersistentObject`



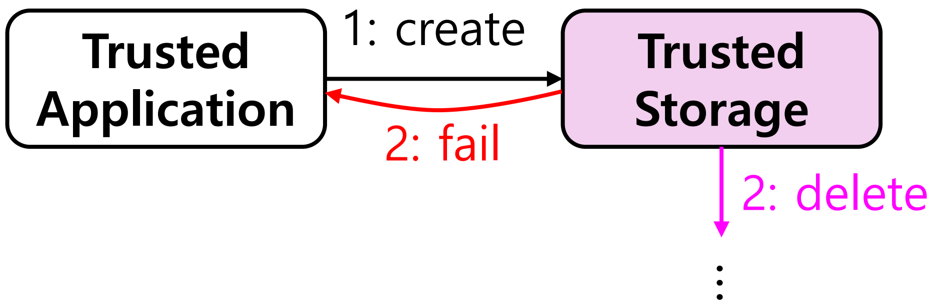
- (2)-1. The storage deletes the old file if an overwrite flag is given.

An example: `TEE_CreatePersistentObject`

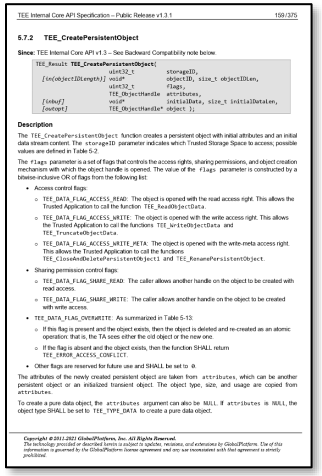


- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

An example: `TEE_CreatePersistentObject`



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.



TEE Internal Core API Specification – Public Release v1.3.1 159/375

5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]             TEE_ObjectHandle* object );
```

Description

The `TEE_CreatePersistentObject` function creates a persistent object with initial attributes and an initial data stream content. The `storageID` parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

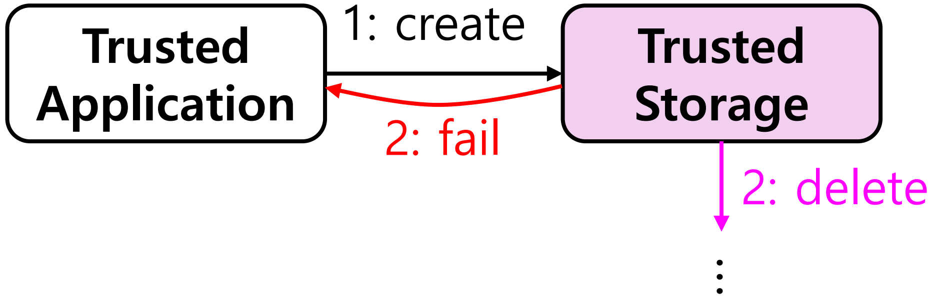
The `flags` parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the `flags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- 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_CloseAndDeletePersistentObject1` 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_OVERWRITE`: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return `TEE_ERROR_ACCESS_CONFLICT`.
- Other flags are reserved for future use and SHALL be set to `0`.

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

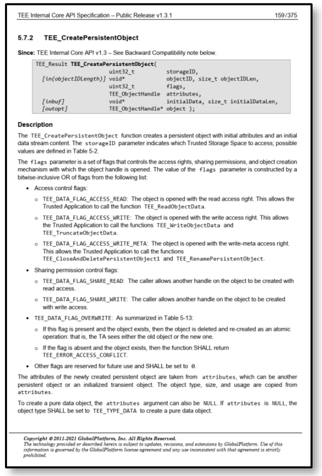
To create a pure data object, the `attributes` argument can also be `NULL`. If `attributes` is `NULL`, the object type SHALL be set to `TEE_TYPE_DATA` to create a pure data object.

An example: TEE_CreatePersistentObject



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

- Trusted storage has the following things:
  - a list of stored files,
  - a counter for object creation,
  - …



TEE Internal Core API Specification – Public Release v1.3.1 159/375

5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]              TEE_ObjectHandle* object );
```

Description

The TEE_CreatePersistentObject function creates a persistent object with initial attributes and an initial data stream content. The storageID parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

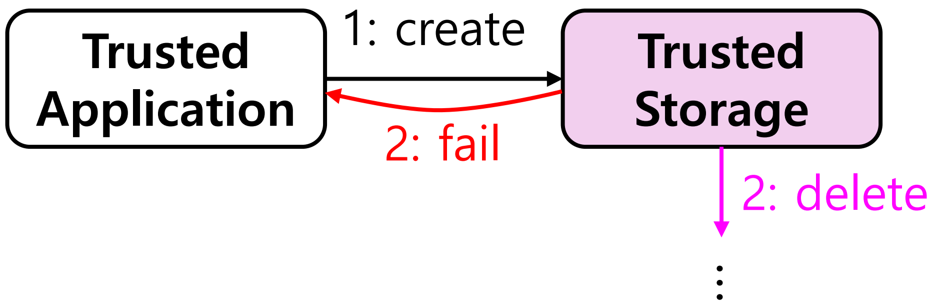
The flags parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the flags parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- 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_CloseAndDeletePersistentObject1 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_OVERWRITE: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return TEE_ERROR_ACCESS_CONFLICT.
- Other flags are reserved for future use and SHALL be set to 0.

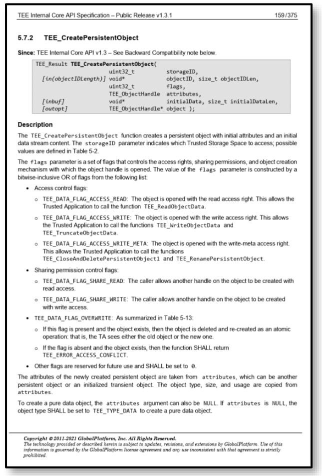
The attributes of the newly created persistent object are taken from attributes, which can be another persistent object or an initialized transient object. The object type, size, and usage are copied from attributes.

To create a pure data object, the attributes argument can also be NULL. If attributes is NULL, the object type SHALL be set to TEE_TYPE_DATA to create a pure data object.

An example: `TEE_CreatePersistentObject`



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.
- Trusted storage has the following things:
  - a list of stored files,
  - a counter for object creation,
  - …



TEE Internal Core API Specification – Public Release v1.3.1 159 / 375

5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]           TEE_ObjectHandle* object );
```

Description

The `TEE_CreatePersistentObject` function creates a persistent object with initial attributes and an initial data stream content. The `storageID` parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

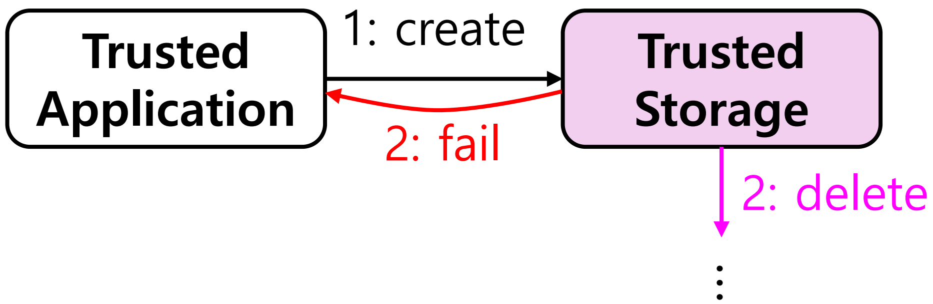
The `flags` parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the `flags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- 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_CloseAndDeletePersistentObject1` 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_OVERWRITE`: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return `TEE_ERROR_ACCESS_CONFLICT`.
- Other flags are reserved for future use and SHALL be set to `0`.

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

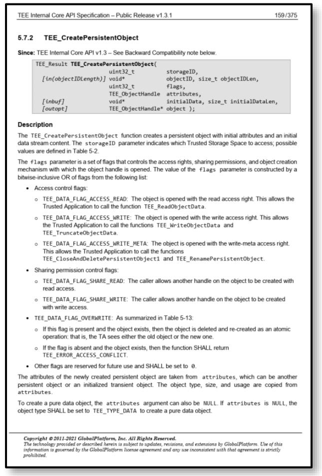
To create a pure data object, the `attributes` argument can also be `NULL`. If `attributes` is `NULL`, the object type SHALL be set to `TEE_TYPE_DATA` to create a pure data object.

An example: `TEE_CreatePersistentObject`



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.
- Trusted storage has the following things:
  - a li
  - a counter for object creation,
  - …

```
class Storage | files : Set{FileName}, counter : Nat, ...
```



TEE Internal Core API Specification – Public Release v1.3.1 159/375

5.7.2 TEE_CreatePersistentObject

Since: TEE Internal Core API v1.3 – See Backward Compatibility note below.

```
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,
  [outopt]              TEE_ObjectHandle*  object );
```

Description

The `TEE_CreatePersistentObject` function creates a persistent object with initial attributes and an initial data stream content. The `storageID` parameter indicates which Trusted Storage Space to access; possible values are defined in Table 5-2.

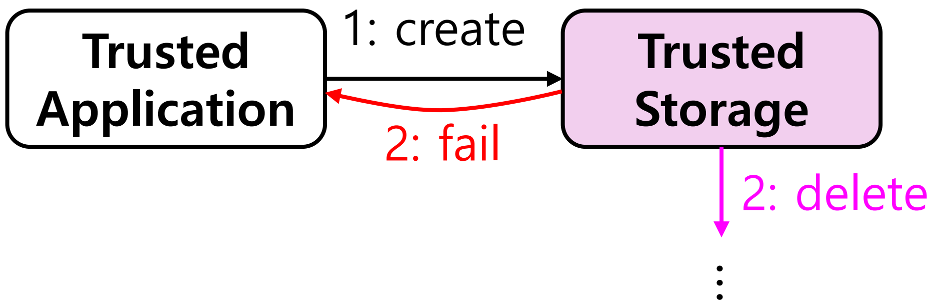
The `flags` parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the `flags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- 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_CloseAndDeletePersistentObject1` 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_OVERWRITE`: As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is, the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return `TEE_ERROR_ACCESS_CONFLICT`.
- Other flags are reserved for future use and SHALL be set to `0`.

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

To create a pure data object, the `attributes` argument can also be `NULL`. If `attributes` is `NULL`, the object type SHALL be set to `TEE_TYPE_DATA` to create a pure data object.

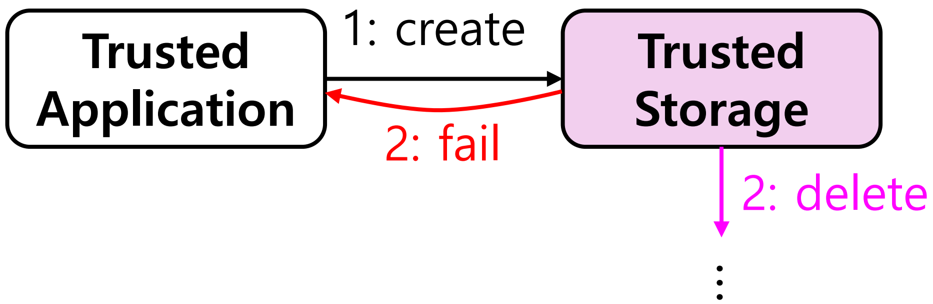
An example: `TEE_CreatePersistentObject`



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

```
crl [create-persistent-overwrite-check]:
   (msg create[METHOD FILE FLAGS HI DATA] from X to SI)
   < PI : PersistObj | file-name : FILE >
   < SI : Storage | status : normal, files : FILES, counter : N >
=> < PI : PersistObj | >
   if overwrite in FLAGS
   then < SI : Storage | counter : N + 2 >
        (msg create[METHOD FILE FLAGS HI DATA N X] from SI to PI)
   else (msg createFail from SI to TK) < SI : Storage | > fi if FILE in FILES .
```

An example: TEE_CreatePersistentObject

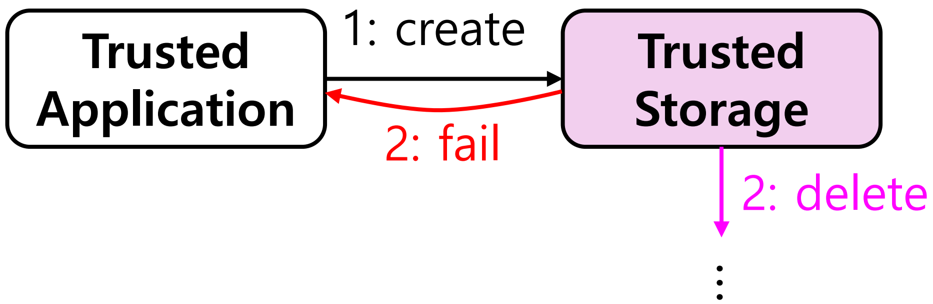


- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

Trusted storage

```
crl [create-persistent-overwrite-check]:
   (msg create[METHOD FILE FLAGS HI DATA] from X to SI)
   < PI : PersistObj | file-name : FILE >
   < SI : Storage | status : normal, files : FILES, counter : N >
=> < PI : PersistObj | >
   if overwrite in FLAGS
   then < SI : Storage | counter : N + 2 >
        (msg create[METHOD FILE FLAGS HI DATA N X] from SI to PI)
   else (msg createFail from SI to TK) < SI : Storage | > fi if FILE in FILES .
```

An example: TEE_CreatePersistentObject



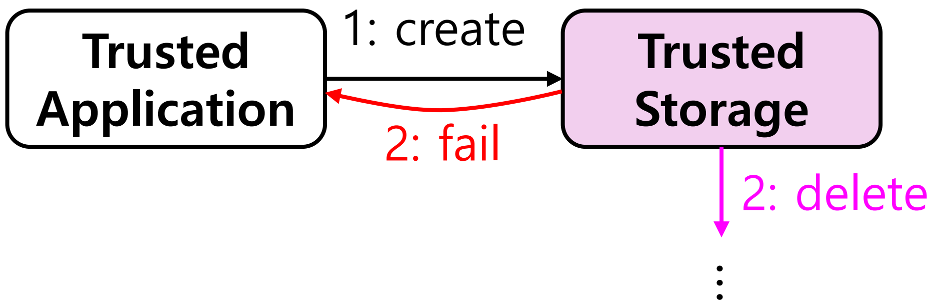
- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

A file creation request message

Trusted storage

```
crl [create-persistent-overwrite-check]:
  (msg create[METHOD FILE FLAGS HI DATA] from X to SI)
  < PI : PersistObj | file-name : FILE >
  < SI : Storage | status : normal, files : FILES, counter : N >
=> < PI : PersistObj | >
   if overwrite in FLAGS
   then < SI : Storage | counter : N + 2 >
        (msg create[METHOD FILE FLAGS HI DATA N X] from SI to PI)
   else (msg createFail from SI to TK) < SI : Storage | > fi if FILE in FILES .
```

An example: `TEE_CreatePersistentObject`



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

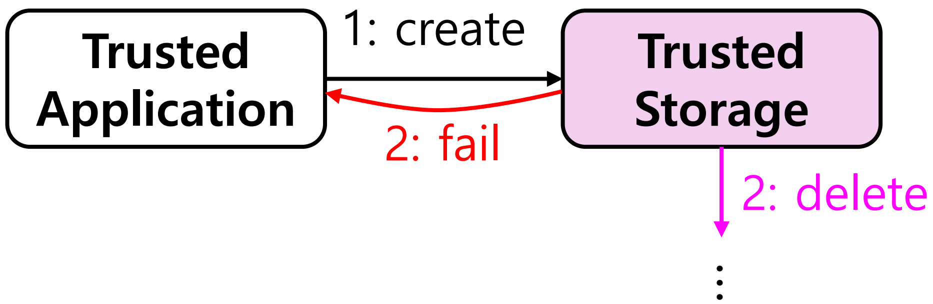
A file creation request message

Trusted storage

```
crl [create-persistent-overwrite-check]:
  (msg create[METHOD FILE FLAGS HI DATA] from X to SI)
  < PI : PersistObj | file-name : FILE >
  < SI : Storage | status : normal, files : FILES, counter : N >
=> < PI : PersistObj | >
   if overwrite in FLAGS
   then < SI : Storage | counter : N + 2 >
        (msg create[METHOD FILE FLAGS HI DATA N X] from SI to PI)
   else (msg createFail from SI to TK) < SI : Storage | > fi if FILE in FILES .
```

Determine if overwrite flag is given

An example: `TEE_CreatePersistentObject`



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

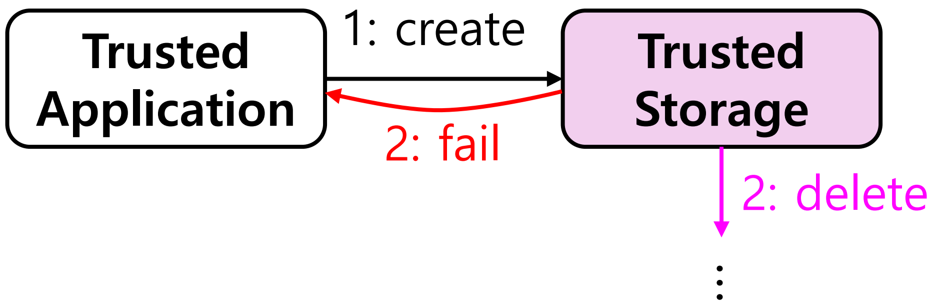
A file creation request message

Trusted storage

```
crl [create-persistent-overwrite-check]:
  (msg create[METHOD FILE FLAGS HI DATA] from X to SI)
  < PI : PersistObj | file-name : FILE >
  < SI : Storage | status : normal, files : FILES, counter : N >
=> < PI : PersistObj | >
  if overwrit
  then < SI : Storage | counter : N + 2 >
       (msg create[METHOD FILE FLAGS HI DATA N X] from SI to PI)
  else (msg createFail from SI to TK) < SI : Storage | > fi if FILE in FILES .
```

Sends a file deletion request message to the old file

An example: TEE_CreatePersistentObject



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

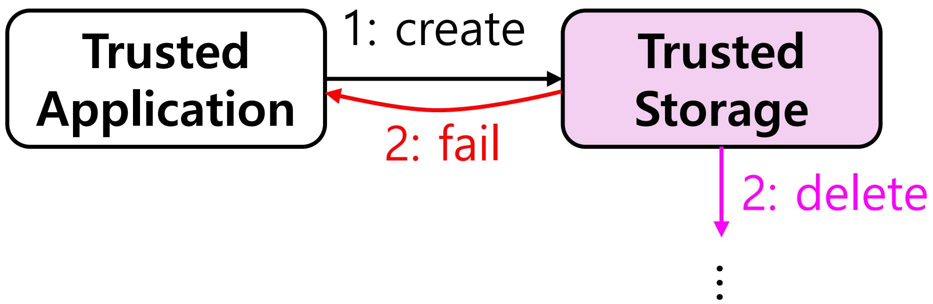
A file creation request message

Trusted storage

```
crl [create-persistent-overwrite-check]:
  (msg create[METHOD FILE FLAGS HI DATA] from X to SI)
  < PI : PersistObj | file-name : FILE >
  < SI : Storage | status : normal, files : FILES, counter : N >
=> < PI : PersistObj | >
  if overwrite in FLAGS
  th                                N + 2 >
  (msg create[METHOD FILE FLAGS HI DATA N X] from SI to PI)
  else (msg createFail from SI to TK) < SI : Storage | > fi if FILE in FILES .
```

If no overwrite flag is given

An example: `TEE_CreatePersistentObject`



- (2)-1. The storage deletes the old file if an overwrite flag is given.
- (2)-2. Otherwise, the storage returns a failure message.

A file creation request message

Trusted storage

```
crl [create-persistent-overwrite-check]:
  (msg create[METHOD FILE FLAGS HI DATA] from X to SI)
  < PI : PersistObj | file-name : FILE >
  < SI : Storage | status : normal, files : FILES, counter : N >
=> < PI : PersistObj | >
   if overwrite in FLAGS
   th                                          N + 2 >
                                               GS HI DATA N X] from SI to PI)
   else (msg createFail from SI to TK) < SI : Storage | > fi if FILE in FILES .
```

If r

Sends a failure message

Formal Specification of TEE APIs

- We specify all API functions of the Trusted Storage API and Cryptographic Operations API.

Formal Specification of TEE APIs

- We specify all API functions of the Trusted Storage API and Cryptographic Operations API.

Trusted Storage API (27/27)

TEE_CreatePersistentObject
TEE_OpenPersistentObject
TEE_RenamePersistentObject
TEE_CloseAndDeletePersistentObject1
TEE_ReadObjectData
TEE_WriteObjectData
…
TEE_CopyObjectAttributes1
TEE_PopulateTransientObject
…

Formal Specification of TEE APIs

- We specify all API functions of the Trusted Storage API and Cryptographic Operations API.

Trusted Storage API (27/27)

TEE_CreatePersistentObject
TEE_OpenPersistentObject
TEE_RenamePersistentObject
TEE_CloseAndDeletePersistentObject1
TEE_ReadObjectData
TEE_WriteObjectData
…
TEE_CopyObjectAttributes1
TEE_PopulateTransientObject
…

Crytographic Operations API (30/30)

TEE_AllocateOperation
TEE_ResetOperation
TEE_SetOperationKey
TEE_CopyOperation
TEE_FreeOperation
TEE_DigestUpdate
…
TEE_MACInit
TEE_MACUpdate
…

Formal Specification of TEE APIs

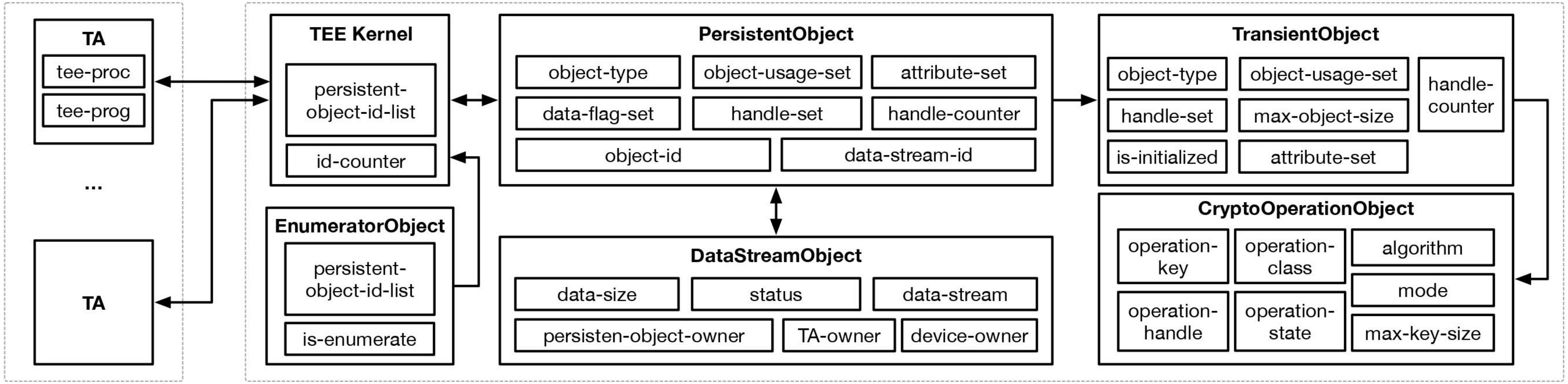
- Our formal model consists of more than 15 objects, and 245 rules.

Formal Specification of TEE APIs

- Our formal model consists of more than 15 objects, and 245 rules.
- We write almost 8K LoC for our specification.

Formal Specification of TEE APIs

- Our formal model consists of more than 15 objects, and 245 rules.
- We write almost 8K LoC for our specification.



Case Study

Case Study

Goal

Demonstrate the effectiveness of our formal model by using it to formally analyze a real-world TEE application.

Case Study

Goal

Demonstrate the effectiveness of our formal model by using it to formally analyze a real-world TEE application.

Settings

- We define the language semantics for TEE applications in Maude.
- We extend our model to run TEE applications using this semantics.

Our Target TEE Application

- As our target TEE application, we choose MQT-TZ [Segarra+20].

Our Target TEE Application

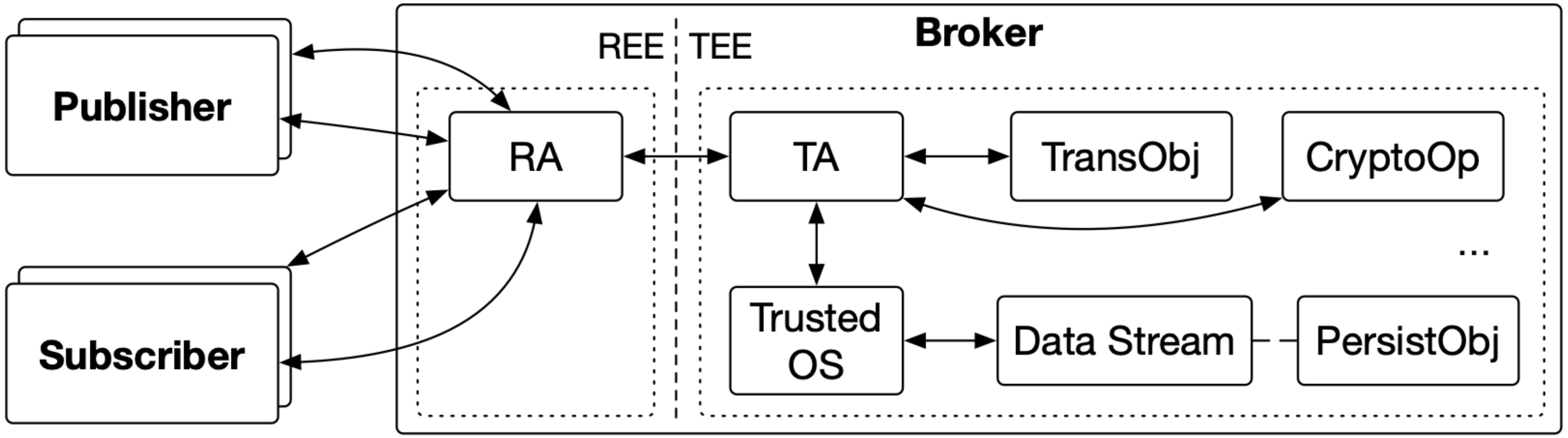
- As our target TEE application, we choose MQT-TZ [Segarra+20].

- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.

Our Target TEE Application

- As our target TEE application, we choose MQT-TZ [Segarra+20].

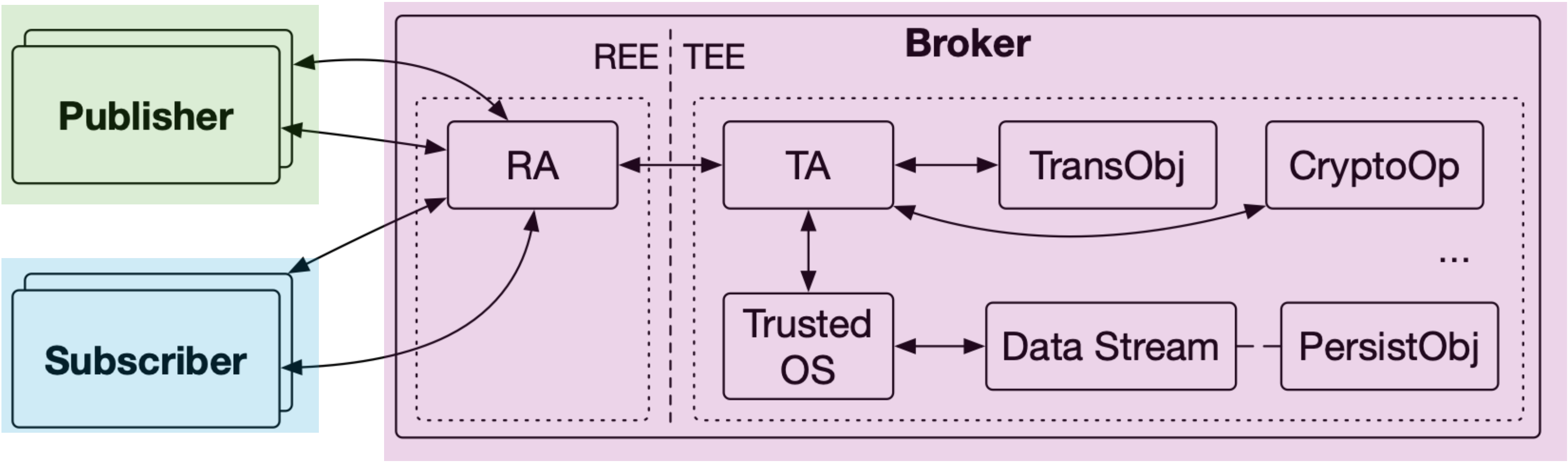
- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.



Our Target TEE Application

- As our target TEE application, we choose MQT-TZ [Segarra+20].

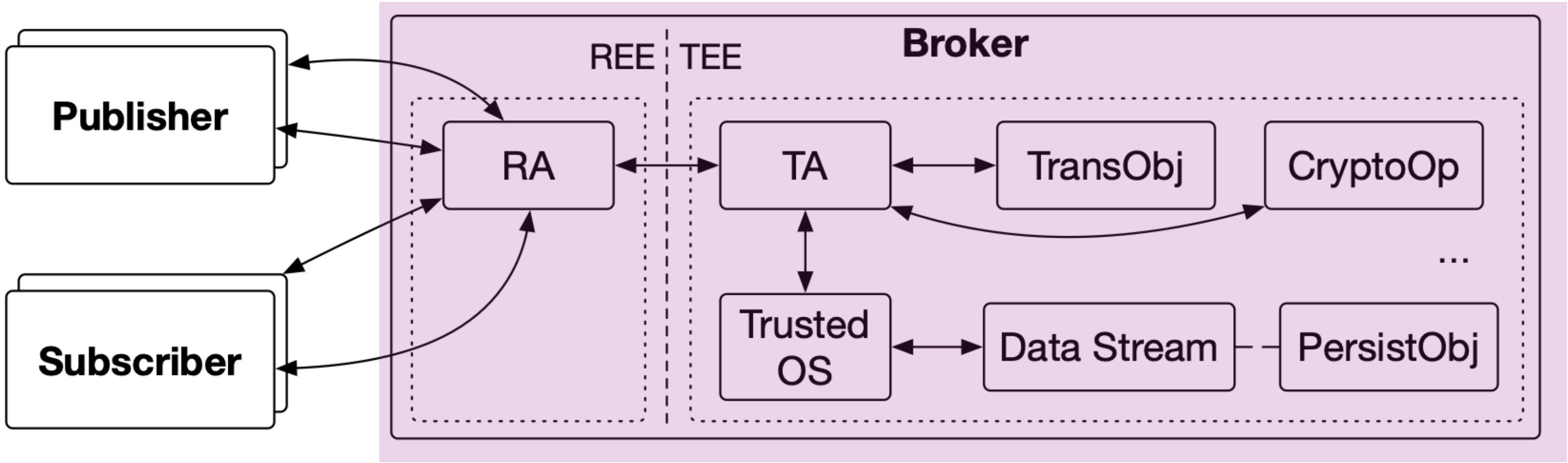
- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.



Our Target TEE Application

- As our target TEE application, we choose MQT-TZ [Segarra+20].

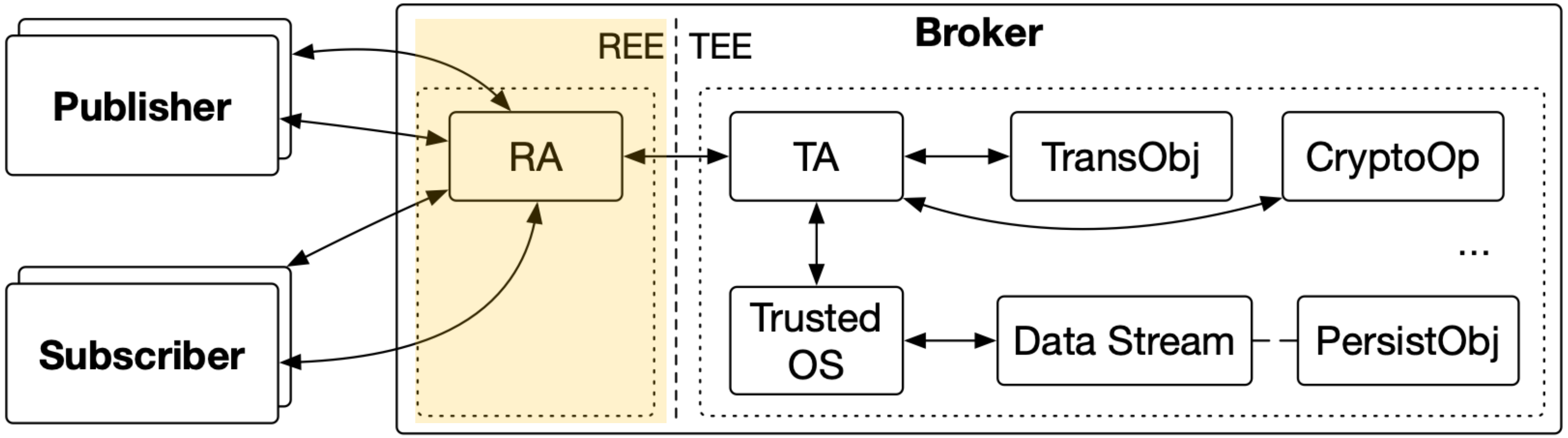
- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.



Our Target TEE Application

- As our target TEE application, we choose MQT-TZ [Segarra+20].

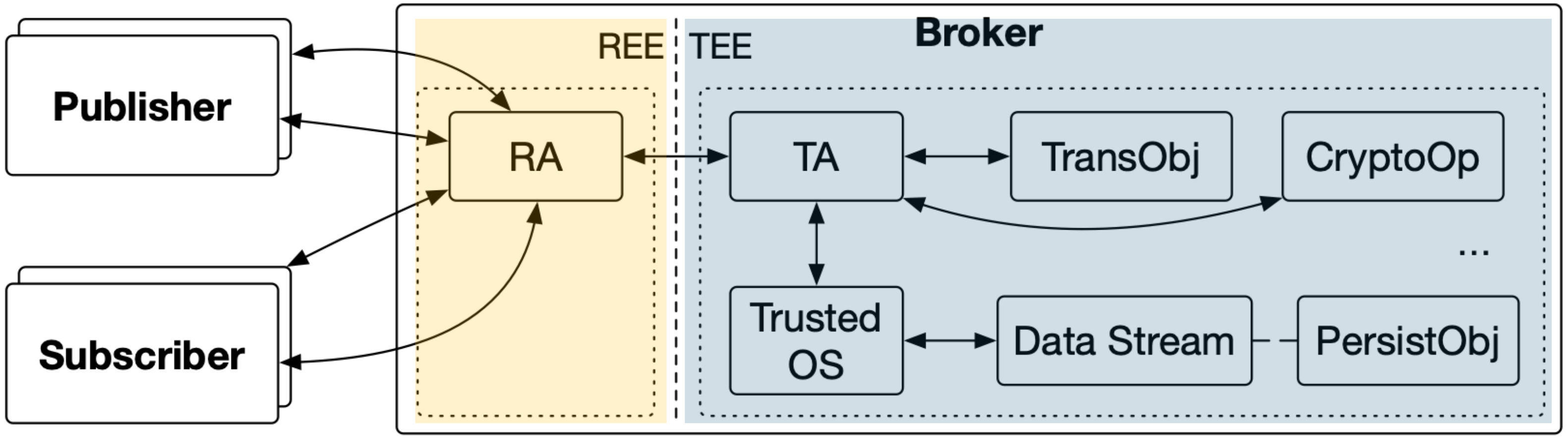
- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.



Our Target TEE Application

- As our target TEE application, we choose MQT-TZ [Segarra+20].

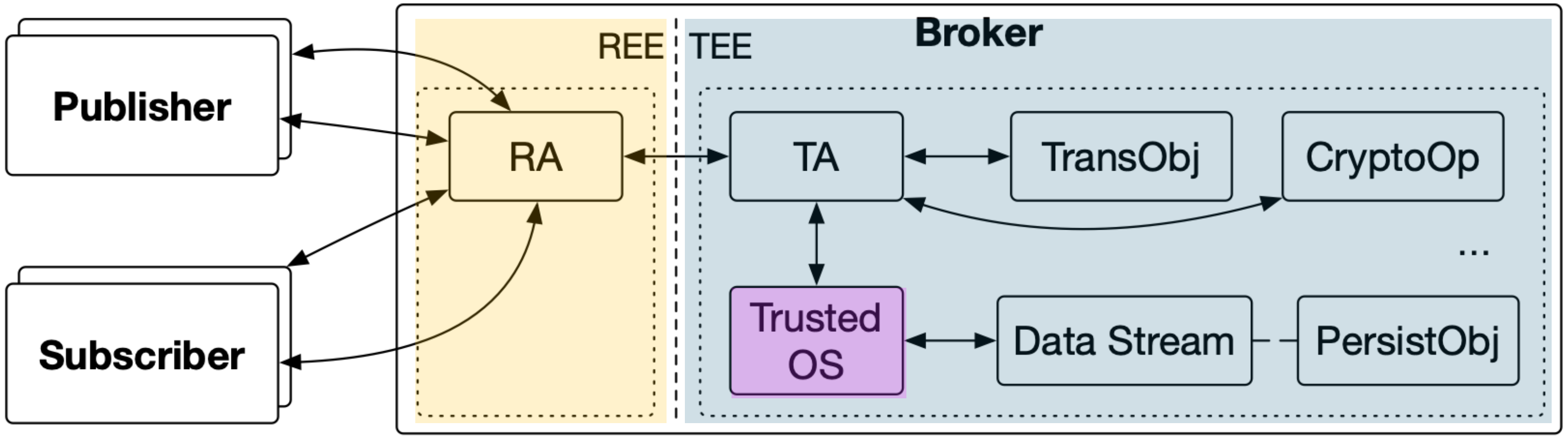
- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.



Our Target TEE Application

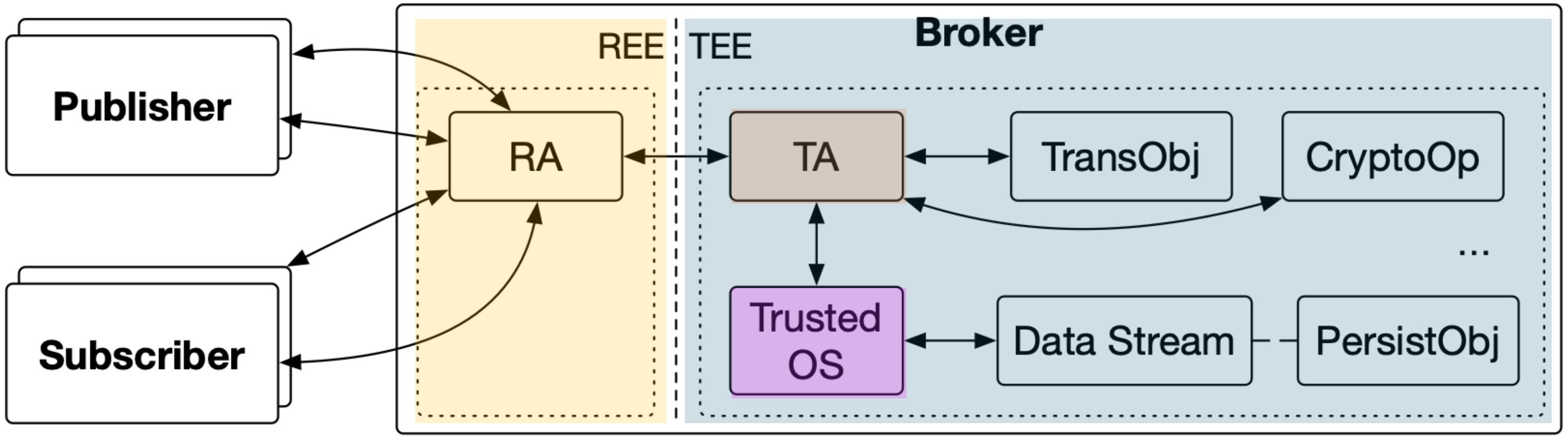
- As our target TEE application, we choose MQT-TZ [Segarra+20].

- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.



Our Target TEE Application

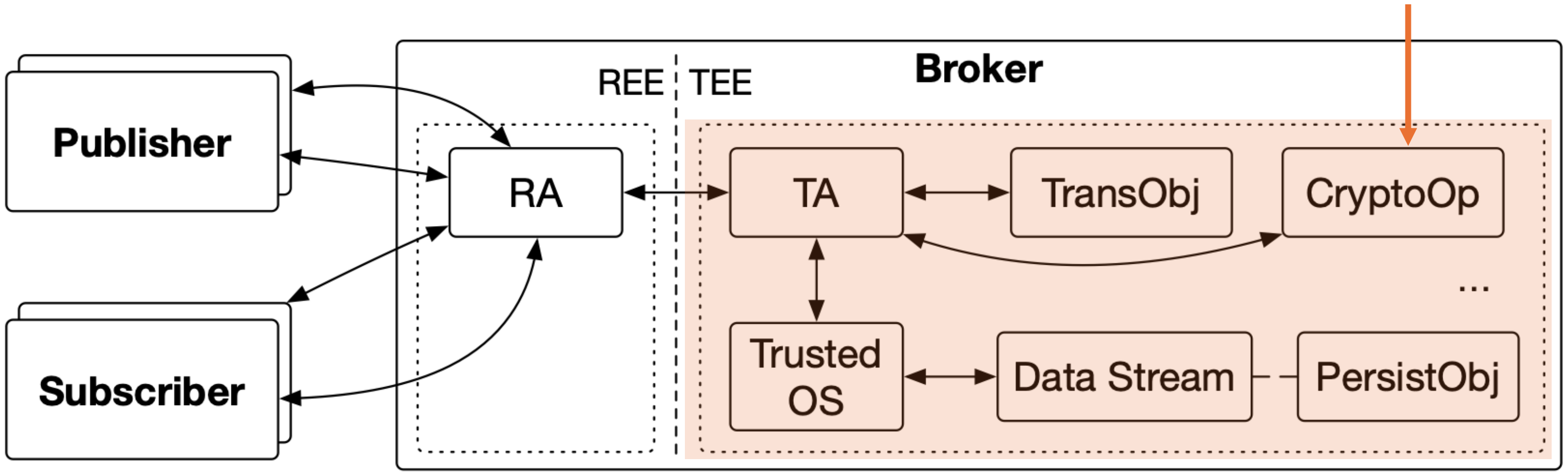
- As our target TEE application, we choose MQT-TZ [Segarra+20].
- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.



Our Target TEE Application

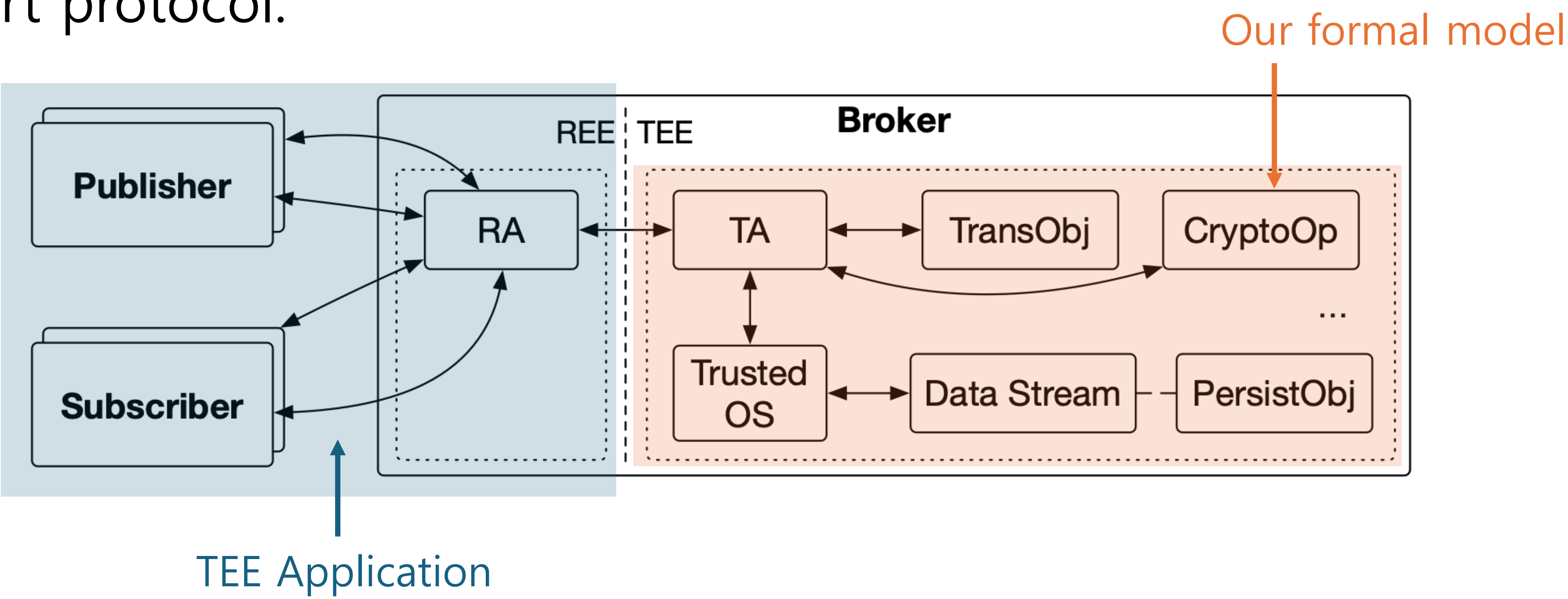
- As our target TEE application, we choose MQT-TZ [Segarra+20].
- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.

Our formal model



Our Target TEE Application

- As our target TEE application, we choose MQT-TZ [Segarra+20].
- MQT-TZ is a TEE-based implementation of a publish-subscribe message transport protocol.



Threat Models

Threat Models

- (1) Memory threat
  - This threat makes brokers to run out of memory.

Threat Models

- (1) Memory threat

  - This threat makes brokers to run out of memory.

- (2) Message modification threat

  - This threat modifies the sender of a message.

Defining Requirements of MQT-TZ

- We define various requirements for MQT-TZ and express them as LTL properties.

Defining Requirements of MQT-TZ

- We define various requirements for MQT-TZ and express them as LTL properties.

Name	Description	LTL Formula
P1	If no memory error occurs in the broker, subscribers eventually receive messages.	$\square \neg memErr.B \rightarrow \square (send.P \rightarrow \lozenge recv.S)$
P2	If the TA panics, subscribers should not receive any messages.	$\square (panic.TA \rightarrow \square \neg recv.S)$
P3	If any memory error occurs in the broker, subscribers should not receive any messages.	$\square (memErr.B \rightarrow \square \neg recv.S)$
P4	When the TA starts running, it should eventually terminate.	$\square (start.TA \rightarrow term.TA)$
P5	If subscribers receive messages from publishers, messages sent from each publisher are in order.	$\square (inQueue.P(a :: b :: c) \rightarrow \lozenge inQueue.S(a :: b :: c))$
P6	The number of tasks handled by the TA cannot exceed five.	$\square (\neg numTaskExceed(5))$

Defining Requirements of MQT-TZ

- We define various requirements for MQT-TZ and express them as LTL properties.

Name	Description	LTL Formula
P1	If no memory error occurs in the broker, subscribers eventually receive messages.	$\square \neg memErr.B \rightarrow \square (send.P \rightarrow \lozenge recv.S)$
P2	If the TA panics, subscribers should not receive any messages.	$\square (panic.TA \rightarrow \square \neg recv.S)$
P3	If any memory error occurs in the broker, subscribers should not receive any messages.	$\square (memErr.B \rightarrow \square \neg recv.S)$
P4	When the TA starts running, it should eventually terminate.	$\square (start.TA \rightarrow term.TA)$
P5	If subscribers receive messages from publishers, messages sent from each publisher are in order.	$\square (inQueue.P(a :: b :: c) \rightarrow \lozenge inQueue.S(a :: b :: c))$
P6	The number of tasks handled by the TA cannot exceed five.	$\square (\neg numTaskExceed(5))$

LTL Model Checking of MQT-TZ

- We perform LTL model checking using Maude.

LTL Model Checking of MQT-TZ

- We perform LTL model checking using Maude.
- We consider three scenarios.
  - `NON` : no threat
  - `OOM` : memory threat
  - `MSG` : message modification threat

LTL Model Checking of MQT-TZ

- We perform LTL model checking using Maude.
- We consider three scenarios.
  - `NON` : no threat
  - `OOM` : memory threat
  - `MSG` : message modification threat

Prop.	Type	Safe?	|S|	Time
P1	NON	$\top$	62	35.7
	MSG	$\top$	148	90.1
	OOM	$\top$	202	144.2
P2	NON	$\top$	62	34.9
	MSG	$\bot$	17	9.1
	OOM	$\top$	532	547.9

Prop.	Type	Safe?	|S|	Time
P3	NON	$\top$	62	35
	MSG	$\top$	148	88.8
	OOM	$\bot$	0.1	0.1
P4	NON	$\top$	62	34.9
	MSG	$\top$	148	88.6
	OOM	$\top$	532	539.3

Prop.	Type	Safe?	|S|	Time
P5	NON	$\top$	62	33.8
	MSG	$\top$	148	86.9
	OOM	$\top$	532	546.7
P6	NON	$\top$	62	34.3
	MSG	$\top$	148	87.9
	OOM	$\top$	532	542.4

LTL Model Checking of MQT-TZ

- We perform LTL model checking using Maude.
- We consider three scenarios.
  - `NON` : no threat
  - `OOM` : memory threat
  - `MSG` : message modification threat

Prop.	Type	Safe?	|S|	Time
P1	NON	$\top$	62	35.7
	MSG	$\top$	148	90.1
	OOM	$\top$	202	144.2
P2	NON	$\top$	62	34.9
	MSG	$\bot$	17	9.1
	OOM	$\top$	532	547.9

Prop.	Type	Safe?	|S|	Time
P3	NON	$\top$	62	35
	MSG	$\top$	148	88.8
	OOM	$\bot$	0.1	0.1
P4	NON	$\top$	62	34.9
	MSG	$\top$	148	88.6
	OOM	$\top$	532	539.3

Prop.	Type	Safe?	|S|	Time
P5	NON	$\top$	62	33.8
	MSG	$\top$	148	86.9
	OOM	$\top$	532	546.7
P6	NON	$\top$	62	34.3
	MSG	$\top$	148	87.9
	OOM	$\top$	532	542.4

LTL Model Checking of MQT-TZ

- We perform LTL model checking using Maude.
- We consider three scenarios.
  - `NON` : no threat
  - `OOM` : memory threat
  - `MSG` : message modification threat

Prop.	Type	Safe?	$\vert S \vert$	Time
	NON	$\top$	62	35.7
P1	MSG	$\top$	148	90.1
	OOM	$\top$	202	144.2
	NON	$\top$	62	34.9
P2	MSG	$\bot$	17	9.1
	OOM	$\top$	532	547.9

Prop.	Type	Safe?	$\vert S \vert$	Time
	NON	$\top$	62	35
P3	MSG	$\top$	148	88.8
	OOM	$\bot$	0.1	0.1
	NON	$\top$	62	34.9
P4	MSG	$\top$	148	88.6
	OOM	$\top$	532	539.3

Prop.	Type	Safe?	$\vert S \vert$	Time
	NON	$\top$	62	33.8
P5	MSG	$\top$	148	86.9
	OOM	$\top$	532	546.7
	NON	$\top$	62	34.3
P6	MSG	$\top$	148	87.9
	OOM	$\top$	532	542.4

LTL Model Checking of MQT-TZ

- We perform LTL model checking using Maude.
- We consider three scenarios.
  - NON : no threat
  - OOM : m
  - MSG : message modification threat

Maude generates counterexamples for violations

Prop.	Type	Safe?	|S|	Time
P1	NON	$\top$	62	35.7
	MSG	$\top$	148	90.1
	OOM	$\top$	202	144.2
P2	NON	$\top$	62	34.9
	MSG	$\bot$	17	9.1
	OOM	$\top$	532	547.9

Prop.	Type	Safe?	|S|	Time
P3	NON	$\top$	62	35
	MSG	$\top$	148	88.8
	OOM	$\bot$	0.1	0.1
P4	NON	$\top$	62	34.9
	MSG	$\top$	148	88.6
	OOM	$\top$	532	539.3

Prop.	Type	Safe?	|S|	Time
P5	NON	$\top$	62	33.8
	MSG	$\top$	148	86.9
	OOM	$\top$	532	546.7
P6	NON	$\top$	62	34.3
	MSG	$\top$	148	87.9
	OOM	$\top$	532	542.4

Analyzing the Violations

- We analyze the counterexample execution paths, generated by Maude.

Analyzing the Violations

- We analyze the counterexample execution paths, generated by Maude.

P2	If the TA panics, subscribers should not receive any messages.	$\square\,(panic.TA \rightarrow \square\,\neg recv.S)$
P3	If any memory error occurs in the broker, subscribers should not receive any messages.	$\square\,(memErr.B \rightarrow \square\,\neg recv.S)$

Prop.	Type	Safe?	|S|	Time	Prop.	Type	Safe?	|S|	Time	Prop.	Type	Safe?	|S|	Time
	NON	$\top$	62	35.7		NON	$\top$	62	35		NON	$\top$	62	33.8
P1	MSG	$\top$	148	90.1	P3	MSG	$\top$	148	88.8	P5	MSG	$\top$	148	86.9
	OOM	$\top$	202	144.2		OOM	$\bot$	0.1	0.1		OOM	$\top$	532	546.7
	NON	$\top$	62	34.9		NON	$\top$	62	34.9		NON	$\top$	62	34.3
P2	MSG	$\bot$	17	9.1	P4	MSG	$\top$	148	88.6	P6	MSG	$\top$	148	87.9
	OOM	$\top$	532	547.9		OOM	$\top$	532	539.3		OOM	$\top$	532	542.4

Analyzing the Violations

- We analyze the counterexample execution paths, generated by Maude.

P2	If Even if the TA panicked, some subscriber receives a message.	$\Box\,(panic.TA \rightarrow \Box\,\neg recv.S)$
P3	If any memory error occurs in the broker, subscribers should not receive any messages.	$\Box\,(memErr.B \rightarrow \Box\,\neg recv.S)$

Prop.	Type	Safe?	|S|	Time
P1	NON	$\top$	62	35.7
	MSG	$\top$	148	90.1
	OOM	$\top$	202	144.2
P2	NON	$\top$	62	34.9
	MSG	$\bot$	17	9.1
	OOM	$\top$	532	547.9

Prop.	Type	Safe?	|S|	Time
P3	NON	$\top$	62	35
	MSG	$\top$	148	88.8
	OOM	$\bot$	0.1	0.1
P4	NON	$\top$	62	34.9
	MSG	$\top$	148	88.6
	OOM	$\top$	532	539.3

Prop.	Type	Safe?	|S|	Time
P5	NON	$\top$	62	33.8
	MSG	$\top$	148	86.9
	OOM	$\top$	532	546.7
P6	NON	$\top$	62	34.3
	MSG	$\top$	148	87.9
	OOM	$\top$	532	542.4

Analyzing the Violations

- We analyze the counterexample execution paths, generated by Maude.

P2	If…	$\square\,(panic.TA \rightarrow \square\,\neg recv.S)$
P3	If… receive any messages.	…$emErr.B$ $\rightarrow \square\,\neg recv.S)$

Even if the TA panicked, some subscriber receives a message.

If memory error occurred in TA, some subscriber still receives a message.

Prop.	Type	Safe?	|S|	Time
P1	NON	⊤	62	35.7
	MSG	⊤	148	90.1
	OOM	⊤	202	144.2
P2	NON	⊤	62	34.9
	MSG	⊥	17	9.1
	OOM	⊤	532	547.9

Prop.	Type	Safe?	|S|	Time
P3	NON	⊤	62	35
	MSG	⊤	148	88.8
	OOM	⊥	0.1	0.1
P4	NON	⊤	62	34.9
	MSG	⊤	148	88.6
	OOM	⊤	532	539.3

Prop.	Type	Safe?	|S|	Time
P5	NON	⊤	62	33.8
	MSG	⊤	148	86.9
	OOM	⊤	532	546.7
P6	NON	⊤	62	34.3
	MSG	⊤	148	87.9
	OOM	⊤	532	542.4

Analyzing the Violations

- The reason is that the broker program cannot distinguish the following three TA status:
  - (1) successful termination,
  - (2) panic,
  - (3) out-of-memory.

Analyzing the Violations

- The reason is that the broker program cannot distinguish the following three TA status:
  - (1) successful termination,
  - (2) panic,
  - (3) out-of-memory.

(2) and (3) → Consider as successful termination

Patching the Bug

- We propose a code-level patch for the broker program to distinguish two error states from successful termination.

Patching the Bug

- We propose a code-level patch for the broker program to distinguish two error states from successful termination.

```
TEEC_Result main(struct test_ctx *ctx, mqttz_client *origin,
                 mqttz_client *dest, mqttz_times *times)
{ ...
  res = TEEC_InvokeCommand(&ctx->sess, TA_REENCRYPT, &op, &ori);



  ...
}
```

Patching the Bug

- We propose a code-level patch for the broker program to distinguish two error states from successful termination.

```
TEEC_Result main(struct test_ctx *ctx, mqttz_client *origin,
                 mqttz_client *dest, mqttz_times *times)
{ ...
  res = TEEC_InvokeCommand(&ctx->sess, TA_REENCRYPT, &op, &ori);
  if (res == TEE_ERROR_OUT_OF_MEMORY || res == TEE_ERROR_TA_DEAD)
  { discardMsg(ctx, origin, dest); }
  ...
}
```

Patching the Bug

- We propose a code-level patch for the broker program to distinguish two error states from successful termination.

```
TEEC_Result main(struct test_ctx *ctx, mqttz_client *origin,
                 mqttz_client *dest, mqttz_times *times)
{ ...
  res = TEEC_InvokeCommand(&ctx->sess, TA_REENCRYPT, &op, &ori);
  if (res == TEE_ERROR_OUT_OF_MEMORY || res == TEE_ERROR_TA_DEAD)
  { discardMsg(ctx, origin, dest); }
  ...
}
```

Successful termination — Out-of-memory — TA panic

Patching the Bug

- After patching, we verify the program again.

Patching the Bug

- After patching, we verify the program again.

Prop.	Type	Safe?	|S|	Time
P1	NON	$\top$	62	35.3
	MSG	$\top$	149	89.9
	OOM	$\top$	203	146.2
P2	NON	$\top$	62	35.1
	MSG	$\top$	149	89.9
	OOM	$\top$	347	294.8

Prop.	Type	Safe?	|S|	Time
P3	NON	$\top$	62	34.8
	MSG	$\top$	149	89.7
	OOM	$\top$	347	285.2
P4	NON	$\top$	62	34.7
	MSG	$\top$	149	89.4
	OOM	$\top$	347	278.5

Prop.	Type	Safe?	|S|	Time
P5	NON	$\top$	62	34.1
	MSG	$\top$	149	87.4
	OOM	$\top$	347	288.6
P6	NON	$\top$	62	34.4
	MSG	$\top$	149	87.9
	OOM	$\top$	347	286.1

Patching the Bug

- After patching, we verify the program again.

Prop.	Type	Safe?	|S|	Time	Prop.	Type	Safe?	|S|	Time	Prop.	Type	Safe?	|S|	Time
P1	NON	⊤	62	35.3	P3	NON	⊤	62	34.8	P5	NON	⊤	62	34.1
	MSG	⊤	149	89.9		MSG	⊤	149	89.7		MSG	⊤	149	87.4
	OOM	⊤	203	146.2		OOM	⊤	347	285.2		OOM	⊤	347	288.6
P2	NON	⊤	62	35.1	P4	NON	⊤	62	34.7	P6	NON	⊤	62	34.4
	MSG	⊤	149	89.9		MSG	⊤	149	89.4		MSG	⊤	149	87.9
	OOM	⊤	347	294.8		OOM	⊤	347	278.5		OOM	⊤	347	286.1

We can confirm that the violated properties are satisfied.

Summary

- We provide a comprehensive formal model for TEE APIs, that can be used in various formal analysis.
- We specify two widely used TEE API categories, Trusted Storage API and Cryptographic Operations API.
- We demonstrate the effectiveness of our model through a case study on formally analyzing a real-world TEE application, MQT-TZ.