Formal Specification of Trusted Execution Environment APIs

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

[1] Pohang University of Science and Technology, Pohang, South Korea
kmbae@postech.ac.kr
[2] Samsung Electronics, Hwasung, South Korea

Abstract. Trusted execution environments (TEEs) have emerged as a key technology in the cybersecurity domain. A TEE provides an isolated environment in which sensitive computations can be executed securely. Trusted applications running in TEEs are developed using standardized APIs that many hardware platforms for TEE adhere to. However, formal models tailored to standard TEE APIs are not well developed. In this paper, we present a formal specification of TEE APIs using Maude. We focus on Trusted Storage API and Cryptographic Operations API, which are foundational to mobile and IoT applications. The effectiveness of our approach is demonstrated through formal analysis of MQT-TZ, an open-source TEE application for IoT. Our formal analysis has revealed security vulnerabilities in the implementation of MQT-TZ, and we patch and confirm its integrity using model checking.

Keywords: Trusted execution environments · formal specification · formal methods · model checking · rewriting logic · Maude

1 Introduction

Trusted execution environments (TEEs) have emerged as a key technology in the cybersecurity of a wide range of software [17]. They provide an isolated program execution environment where sensitive computations can be executed securely, shielding data from both software and hardware attacks. It guarantees the integrity, authenticity, and confidentiality of executed programs and their data. TEE is widely used in security-critical systems such as industrial control systems [5,7], servers [10], mobile security [11], IoT [1,15], etc.

However, the effectiveness of TEEs depends on their proper implementation and use. Inaccuracies or vulnerabilities can compromise the very integrity they seek to maintain; for example, user applications can access an unauthorized region of memory [12], or a kernel can be compromised using a stack-overflow attack [2]. This emphasizes the importance of the formal verification of TEEs. Through rigorous examination and validation, we can ensure the robustness of TEEs, ensuring they operate as intended and providing an additional layer of confidence in their ability to protect critical data.

The standardization of TEE is overseen by Global Platform [8]. Many systems that implement TEE, such as Samsung TEEgris, Trustonic Kinibi, Qualcomm QTEE, etc., adhere to this standard. The standard defines the API for trusted applications (TAs) to handle secure resources, such as memory and storage. These APIs are essential because they provide TEE services to applications running in a TEE. The uniformity of this API specification ensures compatibility across a wide range of applications, even when running on different CPUs.

However, there is an evident deficiency in formal models tailored for TEE specification and its associated APIs. This gap is concerning because without rigorous verification and modeling, the integrity of TEEs could be compromised, potentially exposing vulnerabilities. In this paper, we address this concern by providing a comprehensive formal model of TEE APIs that is explicitly designed for the formal analysis of TEE applications. In this approach, we aim to provide a foundational tool that can serve the diverse spectrum of TEE applications and improve the overall security landscape of software.

The architecture and behavior of Trusted Storage API, precisely defined in the standard [8], is quite complicated. Primarily, it arises from the stringent security requirement that each TA is assigned a dedicated storage, isolated and shielded from other TAs. For example, the function responsible for creating a file in TEE involves multifaceted processes, which is briefly illustrated in Section 3. Such intricacies amplify the difficulty in developing a faithful formal model for TEEs, because of a huge *representation gap* between the informal (standard) specification [8] and a formal model to be developed.

In this paper, we address challenge of the representation gap by leveraging a very expressive modeling language, called Maude [4], which supports powerful object-oriented specification. Since TEE API is mainly specified using objects and their interactions [8], it is appropriate to use such object-oriented modeling approaches to formally specify TEE APIs, making it much easier to develop a comprehensive formal model. We formalize important parts of TEE APIs, namely, Trusted Storage API and Cryptographic Operations API, which are central for trusted applications in mobile and IoT domains.

We demonstrate the effectiveness of our approach for formally analyzing MQT-TZ [20,21], an open-source TEE application that secures the IoT protocol MQTT. We have analyzed several security requirements of the implementation of MQT-TZ and found security vulnerabilities using model checking. We are able to fix a code-level bug and verify through model checking that the fixed program satisfies the previously violated requirements.

This paper is organized as follows. Section 2 provides necessary background on trusted execution environments and Maude. Section 3 presents the formal object-oriented specification of Trusted Storage API in Maude. Section 4 presents the Maude specification of Cryptographic Operations API. Section 5 explains how TEE infrastructures, including trusted applications, can be specified in Maude. Section 6 presents a case study on analyzing various requirements of MQT-TZ and improving the implementation of MQT-TZ using our framework. Section 7 discusses related work. Section 8 presents some concluding remarks.

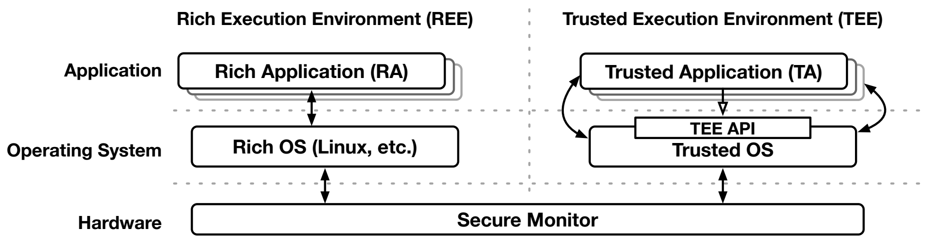

Fig. 1: Overview of the TEE Architecture.

2 Preliminary

Trusted Execution Environments. A trusted execution environment (TEE) uses a physically isolated storage and memory space to protect the security of program codes, executions, sensitive data, and so on. TEE is standardized by Global Platform [8], and many operating systems for TEE (e.g., Samsung TEEgris, Trustonic Kinibi, and Qualcomm QTEE) follow the standard. In particular, the standard defines the API for trusted applications to manage secure resources including memory and trusted storage.

Figure 1 shows the overall architecture of TEE. Trusted applications (TAs) are secure applications running in TEE. In contrast, rich applications (RAs) are normal applications in REE. A trusted OS provides a collection of API functions, specified in the standard document [8], for TAs to perform secure operations. RAs perform secure services by invoking TAs, and the results of such requests are returned to RAs, through a dedicated hardware called a secure monitor.

Maude. Maude [4] is a language and tool for formally specifying and analyzing concurrent systems. A Maude specification consists of: (i) an equational theory (Σ, E) specifying system states as algebraic data types, where Σ is a signature (i.e., declaring sorts, subsorts, and function symbols) and E is a set of equations; and (ii) a set of rewrite rules R of the form $l : t \longrightarrow t'$ **if** *condition*, specifying the system behavior, where l is a label, and t and t' are terms [14].

In Maude, operators are declared with the syntax `op` f `:` $s_1 \dots s_n$ `->` s, where $s_1, ..., s_n$ denote domain sorts and s denotes a range sort. Rewrite rules are declared with the syntax `crl [`l`]:` t `=>` t' `if` *cond* (or, for unconditional rules, `rl [`l`]:` t `=>` t'), where *cond* is a conjunction of equations. Similarly, equations are declared with the syntax `ceq` t `=` t' `if` *cond* (or `eq` t `=` t').

A class declaration `class` C `|` att_1 `:` s_1`,` ...`,` att_n `:` s_n declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An instance of a class C is represented as a term `<` O `:` C `|` att_1 `:` v_1`,` ...`,` att_n `:` v_n `>` of sort `Object`, where O is the object's identifier, and v_i is the value of each attribute att_i. A `subclass` inherits the attributes and rewrite rules of its superclasses. A message is represented as a term of sort `Msg`. A global system state is a term of sort `Configuration` that has the structure of a multiset composed of objects and messages, where multiset union is denoted by juxtaposition (empty syntax).

Maude provides a number of formal analysis methods, including LTL model checking. Maude's LTL model checker checks whether each behavior from an initial state satisfies a linear temporal logic (LTL) formula. A temporal logic formula is constructed by state propositions and temporal logic operators such as `~` (negation), `/\`, `\/`, `[]` ("always"), `<>` ("eventually"), and `U` ("until").

K Framework. K [16] is a rewriting-based framework for defining the semantics of programming languages, in which many languages, including C [6], Java [3], and EVM [9], have been successfully formalized. In K, program states are specified as multisets of cells, called *K configurations.* Each cell represents a component of a program state, such as computations, environments, and stores. Transitions bertween K configurations are defined by rewrite rules.

A computation in K is defined as a $\curvearrowright$-separated sequence of computational tasks. For example, $t_1 \curvearrowright t_2 \curvearrowright \ldots \curvearrowright t_n$ represents the computation consisting of t_1 followed by t_2 followed by t_3, and so on. A task can be decomposed into simpler tasks, and the result of a task is forwarded to the subsequent tasks. E.g., $(5{+}x){*}2$ is decomposed into $x \curvearrowright 5 + \square \curvearrowright \square * 2$, where $\square$ is a placeholder for the result of a previous task. If x evaluates to some value, say 4, then $4 \curvearrowright 5 + \square \curvearrowright \square * 2$ becomes $5 + 4 \curvearrowright \square * 2$, which eventually becomes 18.

The following shows a typical example of K rules for variable lookup, where the k cell contains a computation, *env* contains a map from variables to locations, and *store* contains a map from locations to values:

$$\texttt{lookup} : \langle \underset{v}{\underline{x}} \curvearrowright \ldots\rangle_k \ \langle \ldots x \mapsto l \ldots\rangle_{env} \ \langle \ldots l \mapsto v \ldots\rangle_{store}$$

A horizontal line represents a state change, and "..." indicates irrelevant parts. A cell without horizontal lines is not changed by the rule. By the `lookup` rule, if the first task in k is x, then x is replaced by the value v of x in its location l.

K rules can be translated into ordinary rewrite rules [16]. For example, the lookup rule can be written in Maude as follows, where environments and stores are declared as semicolon-separated multisets of assignments, and and `K`, `ENV`, and `STORE` are Maude variables that match the irrelevant parts:

```
rl [lookup]: k(X ~> K) env(X |-> L ; ENV) store(L |-> V ; STORE)
          => k(V ~> K) env(X |-> L ; ENV) store(L |-> V ; STORE) .
```

3 Formal Specification of Trusted Storage API

Trusted Storage API manages files and cryptographic keys in trusted storage. The architecture and behavior of Trusted Storage API [8] is summarized in Section 3.1. Trusted Storage API is complex due to the security requirement that each TA's storage is isolated and inaccessible to other TAs. We use Maude's object-oriented specification to naturally specify the architecture as a collection of objects (Section 3.2) and the behavior as rewrite rules (Section 3.3).

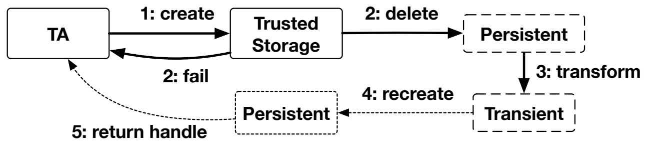

Fig. 2: The flow of TEE_CreatePersistentObject for the case of transformation.

3.1 Overview of Trusted Storage API

In the TEE API standard [8], resources such as files and keys are expressed as objects in an abstract way. A *cryptographic object* contains *attributes*, which are data used to store key material in a structured way. A *persistent object* represents a file associated with a *data stream* in its storage, and may also be a cryptographic object with attributes. A *transient object* represents an object with attributes in memory, but no data streams. *Object handles* are references that identify a particular object and contain access rights information.

There are a total of 26 functions in Trusted Storage API. The persistent API functions can create, rename, and delete persistent objects and their data streams. The data stream API functions can read, write, truncate, or seek data from persistent objects. The transient API functions can allocate and deallocate transient objects, set, reset, or copy cryptographic keys to the objects, or generate random keys. In addition, these functions can open object handles for persistent and transient objects, respectively.

To illustrate the complexity of Trusted Storage API, consider the function TEE_CreatePersistentObject, which creates a persistent object and returns the object handle. It first checks if a persistent object with the same name exists. Then, depending on the overwrite access flag, the operation either fails, or the object is deleted and recreated. A new persistent object can be created either as a cryptographic object or as a pure data object (without attributes). In the former case, attributes can be taken from another cryptographic object, or a transient object can be transformed to the persistent object. We describe the execution flow of transformation when a persistent object already exists, in Figure 2. The dashed box denotes deletion, and the dotted box represents creation.

3.2 Representing Trusted Storage Objects in Maude

Trusted Storage API can naturally be formalized in an object-oriented style. A cryptographic object is modeled as an instance of the class `CryptoObj`, where the attributes `type`, `max-size`, and `usages` denote the type, maximum size, and usages of a cryptographic key to be created, respectively; and `attributes` denotes cryptographic attributes.

```
class CryptoObj | type : Type,       max-size : Nat,   usages : Set{Usage},
                  attributes : Set{CryptoAttribute} .
```

A persistent object is modeled as an instance of the class `PersistObj`, where the attribute `file-name` denotes the name of its file, and `data-stream` denotes the associated data stream. Similarly, a transient object is modeled as an instance of the class `TransObj`, where `initialized` indicates whether the object is initialized. Both classes are declared as subclasses of `CryptoObj`, because they are both cryptographic objects according to the standard [8].

```
class PersistObj | file-name : FileName,   data-stream : List{Data} .
class TransObj   | initialized : Bool .
subclass TransObj PersistObj < CryptoObj .
```

A handle is represented as an instance of a subclass of the class `Handle`, where `oid` is the object that it points to. In particular, an object handle is represented as instances of the subclass `ObjHandle`, where `flags` contains data access flags.

```
class Handle | oid : Oid .          class ObjHandle | flags : Set{DataAccessFlag} .
subclass ObjHandle < Handle .
```

The storage of each TA is modeled as an instance of the class `Storage`, where `status` denotes its status, `files` denotes the file names in the storage, and `counter` denotes a counter for creating a new identifier.

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

The kernel of each TA is modeled as an instance of the class `TAKernel`, where `status` denotes its status, `storage` denotes its storage, `counter` denotes a counter for creating a new identifier, and `api-call` denotes the status of an API call. The status of a TA can be `normal`, `outOfMemory`, or `panic`.

```
class TAKernel | status : AppStatus,     storage : Oid,
                 counter : Nat,          api-call : CallStatus .
```

We represent an API function call as $f(vl)$ `#` n of sort `CallStatus`, where f is a function identifier, vl is the call parameters, and (optional) n denotes the step of the call. The return of the call is represented as `return(`f`,`rl`)`, where rl denotes the return values. We use `return(`f`)` if there are no return values.

The interactions between the objects are represented as the messages of the form `msg` r`[`vl`] from` *Sender* `to` *Receiver*, where r is the name of a request and vl is a list of arguments for the request. We use `msg` r `from` *Sender* `to` *Receiver* for the request with no arguments. For example, `msg getStatus from TK to SI` represents a request message from the TA kernel `TK` to its associated storage `SI` for returning the status with no arguments.

The following example shows a TA and its associated storage, a transient object and its object handle, and a persistent object named `file1`.

```
< tk : TAKernel | status : normal, id-counter : 1, storage : so, ... >
< oh : ObjHandle | oid : to, flags : empty >
< so : Storage | status : normal, files : fileName('file1), counter : 1 >
< to: TransObj | type : rsaKeyPair, max-size : 15, usages : decrypt >
< po : PersistObj | file-name : fileName('file1), type : rsaKeyPair, ... >
```

3.3 Specifying Trusted Storage API Behaviors

Specification of TEE_ReadObjectData. This function takes a single parameter, a handle to a persistent object for data reading. A TA first checks the storage status by sending a message `getStatus` to an associated storage. When the storage receives `getStatus`, it returns its `status` using a message `retStatus`.

```
rl [read-object-data-get-storage-status]:
   < TK : TAKernel | api-call : readObjData(HI), storage : SI >
=> < TK : TAKernel | api-call : readObjData(HI) # 1 > (msg getStatus from TK to SI)
.
rl [return-storage-status]:
   < SI : Storage | status : STATUS > (msg getStatus from TK to SI)
=> < SI : Storage | > (msg retStatus[STATUS] from SI to TK) .
```

If the storage status is normal, the TA sends a message `read` to the handle to request data reading. Otherwise, it returns the storage status.

```
rl [read-object-data-storage-status-check]:
   (msg retStatus[STATUS] from SI to TK)
   < TK : TAKernel | api-call : readObjData(HI) # 1 >
=> if STATUS == normal then
     < TK : TAKernel | api-call : readObjData(HI) # 2 > (msg read from TK to HI)
   else < TK : TAKernel | api-call : return(readObjData, STATUS) > fi .
```

When the handle receives `read` and has the flag `accessRead`, it reads the first data from the data stream of the persistent object. The data is returned to the TA using a message `retData` and the TA returns the received data.

```
rl [read-object-data-from-persist]:
   < HI : ObjHandle | oid : PI, flags : (accessRead, FLAGS) >
   < PI : PersistObj | data-stream : DATA :: STREAM > (msg read from TK to HI)
=> < PI : PersistObj | data-stream : STREAM > (msg retData[DATA] from HI to TK)
   < HI : ObjHandle | > .

rl [read-object-data-success]:
   (msg retData[DATA] from HI to TK)
   < TK : TAKernel | api-call : readObjData(HI) # 2 >
=> < TK : TAKernel | api-call : return(readObjData, DATA) > .
```

Specification of TEE_CreatePersistentObject. Due to the page limit, we explain the rules used to specify the behavior in Figure 2. This function takes five parameters: file name, access flags, a handle to another transient or persistent object, initial data, and an optional handle. A TA determines the method for creating a persistent object and sends a creation request to an associated storage.

```
rl [create-persistent-determine-case]:
   < TK : TAKernel | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT),
                     storage : SI >
=> < TK : TAKernel | api-call : createPersistent(FILE, FLAGS, HI, DATA, OPT) # 1 >
   mkCreationMsg(FILE, FLAGS, HI, DATA, OPT, SI, TK) .
```

The mkCreationMsg function determines the creation method and constructs a create message, where the first argument denotes the method id. If the handle is null, the message is for creating a pure persistent object. If both the handle and optional handle are not null, the message is for creating a persistent object. Otherwise, it's for transforming a transient object into a new persistent object.

```
op mkCreationMsg : FileName Set{DataAccessFlag} HandleId Data HandleId
                   Oid Oid -> Configuration .
eq mkCreationMsg(FILE, FLAGS, null, DATA, OPT, SI, TK)
 = (msg create[pure FILE FLAGS null DATA] from TK to SI) .

ceq mkCreationMsg(FILE, FLAGS, HI, DATA, OPT, SI, TK)
  = if OPT == null
    then (msg create[transform FILE FLAGS HI DATA] from TK to SI)
    else (msg create[persist FILE FLAGS HI DATA] from TK to SI) fi if HI =/= null .
```

When the storage receives the create message, it checks the existence of a persistent object with the same name from the storage. If the object exists and the access flags contain the overwrite flag, it proceeds by sending the create message to the persistent object. Otherwise, it informs TA with createFail.

```
crl [create-persist-overwrite-check]:
   (msg create[METHOD FILE FLAGS HI DATA] from TK 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 TK] from SI to PI)
   else (msg createFail from SI to TK) < SI : Storage | > fi if FILE in FILES .
```

When the persistent object receives the create message with the transform method, it transforms the transient object into a persistent object, opens a new object handle, and deletes itself. Then, the handle is sent to the TA through the message createSuccess. The function newOid is used to create a fresh identifier.

```
crl [create-persist-transform]:
   (msg create[transform FILE FLAGS HI DATA N TK] from SI to PI)
   < HI : ObjHandle | oid : OI >
   < OI : TransObj | type : TYPE, usages : USAGES, max-size : M,
                     attributes : ATTRS >
   < PI : PersistObj | file-name : FILE >
=> < NEW-HI : ObjHandle | oid : NEW-PI, flags : FLAGS >
   < NEW-PI : PersistObj | type : TYPE, usages : USAGES, max-size : M,
                           attributes : ATTRS, data-stream : DATA,
                           file-name : FILE >
   (msg createSuccess[NEW-HI] from NEW-PI to TK)
if NEW-HI := newOid(N, SI) /\ NEW-PI := newOid(N + 1, SI) .
```

When the TA receives a createSuccess message with an object handle, it returns the handle. If receiving createFail or detecting insufficient memory, it returns a corresponding error.

```
rl [create-persist-success]: (msg createSuccess[HI] from PI to TK)
   < TK : TAKernel | status : normal, api-call : createPersistent(VL) >
=> < TK : TAKernel | api-call : return(createPersistent, HI) > .

rl [create-persist-fail]: (msg createFail from SI to TK)
   < TK : TAKernel | status : normal, api-call : createPersistent(VL) >
=> < TK : TAKernel | api-call : return(createPersistent, errorAccessConflict) > .

rl [create-persist-mem-err]:
   < TK : TAKernel | app-status : outOfMemory, api-call : createPersistent(VL) >
=> < TK : TAKernel | api-call : return(createPersistent, errorOutOfMemory) > .
```

4 Formal Specification of Cryptographic Operations API

Cryptographic Operations API handles cryptographic algorithms by managing operation states. Cryptographic Operations API is also quite complex due to the internal operation states. This section shows that these difficulties can be effectively dealt with using Maude's object-oriented specification.

4.1 Overview of Cryptophic Operations API

A *cryptographic operation* abstracts a cryptographic process. It has an operation state such as *initial*, *active*, or *extract*. An *operation handle* is a reference to a cryptographic operation. Each handle has a handle state, which is defined by whether a key is set, an operation is initialized, and data can be extracted.

The API provides a total of 30 functions for various types of cryptographic primitives and schemes, including symmetric ciphers, authenticated encryptions, and key derivations. In addition, the generic operation API functions support the operations common to all types. These functions can allocate, free, reset cryptographic operations, and set cryptographic key.

To illustrate the complexity of Cryptographic Operations API, consider the state diagram of symmetric ciphers, described in Figure 3. The operation can be started either with or without key (`KEY_SET` or **`not`** `KEY_SET`). If it has no key, `TEE_SetOperationKey` is used to set a key. Otherwise, it is initialized (`INIT`) by `TEE_CipherInit`. The operation can run the algorithm with `TEE_CipherUpdate`. After performing the operation, `TEE_FreeOperation` can be used to deallocate the operation or `TEE_CipherDoFinal` is used to finish and reset the operation.

4.2 Representing Cryptographic Operations in Maude

Cryptographic operations can naturally be modeled in an object-oriented style. We model cryptographic operations as instances of class `CryptoOp`. The attribute `attributes` denotes a set of `CryptoAttribute`, `max-size` is the maximum size of a key to use, and `algorithm` is the identifier of an algorithm to operate. The attributes `mode`, `state`, and `opclass` denote the mode, state, and class of the operation, respectively, and `acc-data` is a list of `Data` it holds.

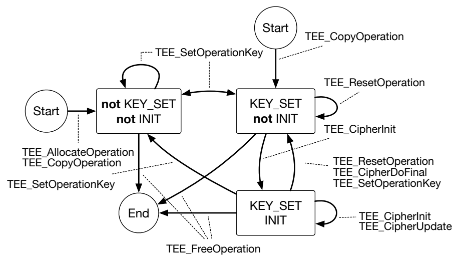

Fig. 3: Symmetric cipher operation.

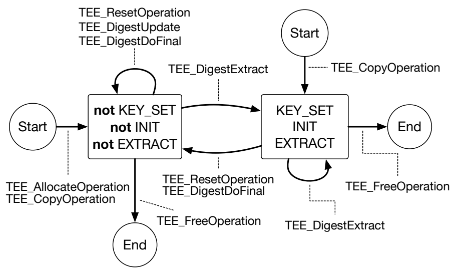

Fig. 4: Message digest operation.

```
class CryptoOp | attributes : Set{CryptoAttribute},  max-size : Nat,
                 algorithm : Algorithm, mode : Mode, state : State,
                 opclass : OpClass,     acc-data : List{Data} .
```

Operation handles are represented as instances of the class `OpHandle`, which extends `Handle`. The attribute `state` is a handle state and `key-material-set` denotes whether cryptographic key materials are set to the operation.

```
class OpHandle | state : HandleState, key-material-set : Bool .
subclass OpHandle < Handle .
```

Specification of TEE_AllocateOperation. This function takes three parameters: an algorithm identifier, a mode, and the maximum key size. A TA first checks whether the algorithm and mode are compatible using the `compatible` function. If valid, it creates a new cryptographic operation, and opens and returns an operation handle. The function `getClass` is used to retrieve the algorithm class.

```
crl [allocate-operation-success]:
    < TK : TAKernel | api-call : allocOperation(ALGO, MODE, MAXSIZE),
                      status : normal, id-counter : N >
 => < TK : TAKernel | api-call : return(allocOperation, HI), id-counter : N + 2 >
    < HI : OpHandle | oid : OI, state : noKeyNotInit, key-material-set : false >
    < OI : CryptoOp | attributes : empty, max-size : MAXSIZE, handle : HI,
                      algorithm : ALGO, mode : MODE, opclass : getClass(ALGO),
                      acc-data : nil, state : initial >
if compatible(ALGO, MODE) /\ OI := newOid(N, TK) /\ HI := newOid(N + 1, TK) .
```

If the algorithm and mode are not compatible or insufficient memory is detected, the TA returns a corresponding error, specified by the following rules:

```
crl [allocate-operation-params-err]:
    < TK : TAKernel | api-call : allocOperation(ALGO, MODE, MAXSIZE) >
 => < TK : TAKernel | api-call : return(allocOperation, errorNotSupported) >
if not compatible(ALGO, MODE) .

rl [allocate-operation-memory-err]:
   < TK : TAKernel | status : outOfMemory, api-call : allocOperation(VL) >
=> < TK : TAKernel | api-call : return(allocOperation, errorOutOfMemory) > .
```

Specification of TEE_ResetOperation. A TA creates a `resetOp` message to reset a cryptographic operation. If the cryptographic operation receives a request and its key materials are set, it resets the operation state using the `resetState` function, clears the data, and notifies the TA using a message `finishResetOp`. The function `resetState` updates the state to initial if the state is active.

```
rl [reset-operation-request-reset]:
   < TK : TAKernel | api-call : resetOperation(HI) > < HI : OpHandle | oid : CI >
=> < TK : TAKernel | >  < HI : OpHandle | > (msg resetOp[HI] from TK to CI) .

rl [reset-operation-finish-reset]:
   < CI : CryptoOp | state : STATE > (msg resetOp[HI] from TK to CI)
   < HI : OpHandle | oid : CI, key-material-set : true >
=> < CI : CryptoOp | acc-data : nil, state : resetState(STATE) >
   < HI : OpHandle | > (msg finishResetOp from CI to TK) .

rl [reset-operation-success]:   (msg finishResetOp from CI to TK)
                 < TK : TAKernel | api-call : resetOperation(VL) >
=> < TK : TAKernel | api-call : return(resetOperation) > .
```

Specification of TEE_CipherUpdate. This function takes two parameters: an operation handle and input data. A TA creates a message `reqCipher` to request data encryption or decryption. When a cryptographic operation receives the message and key materials are set, it checks whether the operation can succeed using the `cipherSuccess` function. If successful, the operation runs the algorithm with `runAlgo` and returns a result to the TA using the `finishCipher` message. Otherwise, it reports failure using the `failCipher` message.

```
rl [cipher-update-request-cipher]:     < HI : OpHandle | oid : CI >
                < TK : TAKernel | api-call : cipherUpdate(HI, DATA) >
=> < TK : TAKernel | > < HI : OpHandle | > (msg reqCipher[HI DATA] from TK to CI) .

rl [cipher-update-try-cipher]:
   (msg reqCipher[HI DATA] from TK to CI)
   < HI : OpHandle | key-material-set : true >
   < CI : CryptoOp | attributes : ATTRS, algorithm : ALGO, mode : MODE,
                     opclass : CLASS, state : STATE >
=> < CI : CryptoOp | > < HI : OpHandle | >
   if cipherSuccess(ALGO, MODE, ATTRS, CLASS, STATE, DATA) then
     (msg finishCipher[runOp(ALGO, MODE, ATTRS, DATA)] from CI to TK)
   else (msg failCipher from CI to TK) fi .
```

When the TA receives the encrypted or decrypted data from `cipherSuccess`, it returns the data. If receiving `failCipher`, it goes to `panic`.

```
rl [cipher-update-success]:     (msg cipherSuccess[VALUE] from CI to TK)
   < TK : TAKernel | api-call : cipherUpdate(VL) >
=> < TK : TAKernel | api-call : return(cipherUpdate, VALUE) > .

rl [cipher-update-panic]:
   < TK : TAKernel | api-call : cipherUpdate(VL) > (msg failCipher from CI to TK)
=> < TK : TAKernel | status : panic > .
```

5 Formal Specification of a TEE Infrastructure

5.1 Representing Rich and Trusted Applications in Maude

Thanks to the K semantics, we can model RA and TA to run programs, written in any programming language. Applications are represented as instances of the following class `App`, where `prog` denotes a program and `proc` is a K configuration for the program execution. RAs and TAs are modeled as instances of the classes `RA` and `TA`, respectively. Both classes inherit `App` but `TA` also inherits `TAKernel`.

```
class App | prog : Program, proc : KConfig .

class RA .                      class TA .
subclass RA < App .             subclass TA < App TAKernel .
```

In this paper, we define K rewrite rules for a subset of the C language, including function calls, variables, assignments, loops, and conditional statements. As mentioned in Section 2, the K semantics can be written in Maude.

For TEE API function calls, we use `TAKernel` to handle them. When a TEE API function `FUNC` is called with parameters `VL`, a TA pushes the call to `api-call` and adds a task `$wait(`f`)`, representing the task waiting for the function f. Then, a `TAKernel` handles the call as explained in Sections 3 and 4. The `isTeeApi` function is used to check whether a function is a TEE API.

```
crl [tee-api-call]:
    < TI : TA | proc : (k(FUNC(VL)    ~> K) KS) >
 => < TI : TA | proc : (k($wait(FUNC) ~> K) KS), api-call : FUNC(VL) >
if isTeeApi(FUNC) .
```

After the `TAKernel` handles the call, the TA assigns the return values to the function's output variables. We use `$out(`xl`)` to denote output variables xl. The `makeRetStmt` function is used to create statements for assigning variables.

```
crl [tee-api-call-return]:
    < TI : TA | proc : (k($wait(FUNC) ~> $out(XL) ~> K) KS),
                api-call : return(FUNC, VL) >
 => < TI : TA | proc : (k(STMT ~> K) KS), api-call : noCall >
if isTeeApi(FUNC) /\ STMT := makeRetStmt(VL, XL) .
```

5.2 Representing Execution Environments

We represent the two separated execution environments as a pair `{`S_R`} | [`S_T`]`, where S_R contains `RA`s and S_T contains `TA`s, together with objects and messages introduced in Sections 3 and 4. Trusted OS is represented as an instance of the class `TrustedOS`, where `sess` is a map from `SessionId` to `Oid`. Sessions are communication channels between RA and TA.

```
class TrustedOS | sess : Map{SessionId,Oid} .
```

We specify the communications between an RA and a TA using Maude rules. The RA calls the TA using a secure monitor call (SMC). We define its semantic using the following rule. A message `smcReq` represents an SMC and the function `makeSmcArgs` makes SMC arguments.

```
crl [invoke-ta]:
      < RI : RA | proc : (k(FUNC(VL) ~> K) KS) >
   => < RI : RA | proc : (k($wait(FUNC) ~> K) KS) > smcReq(ARGS)
if isInvokeFunc(FUNC) /\ ARGS := makeSmcArgs(RI, FUNC, VL) .
```

The secure monitor accepts the SMC request by transferring the message `smcReq` from REE to TEE. Later, it gets a result from TEE through a message `smcRet` and finishes the request by transferring the message to REE.

```
rl [accept-smc-request]: {REE smcReq(ARGS)} | {TEE} => {REE} | {TEE smcReq(ARGS)} .
rl [return-smc-request]: {REE} | {TEE smcRet(ARGS)} => {REE smcRet(ARGS)} | {TEE} .
```

We define the behavior of a trusted OS when receiving `smcReq`. The OS invokes a target TA using an `invkTa` message. The function `getTargetTa` is used to extract the target TA from SMC arguments and `getRequestor` is used to get the RA's identifier.

```
crl [accept-smc-request]:
    < OS : TrustedOS | sess : SM > smcReq(ARGS)
 => < OS : TrustedOS | > invkTa(TI, RI, ARGS)
if RI := getRequestor(ARGS) /\ TI := getTargetTa(ARGS, SM) .
```

When the target TA receives `invkTa` and is not running, it executes a program using the function `run`. For example, `run`(p, f, vl) executes the function f of a program p with arguments vl. The functions `getFunc` and `getParams` are used to get a function identifier and call parameters from SMC arguments.

```
crl [handle-invoke-ta]:
    < TI : TA | proc : none, prog : P > invkTa(TI, RI, ARGS)
 => < TI : TA | proc : run(P, F, VL) > invkTa(TI, RI, ARGS)
if F := getFunc(ARGS) /\ VL := getParams(ARGS) .
```

After the execution, the TA gets a result from `proc` using the function `getRes` and creates an `invkTaRet` message. Then, the trusted OS creates an `smcRet` message for sending the result to the secure monitor, which is transferred to REE. The function `finished` checks whether the process is finished.

```
crl [handle-invoke-ta-finish]:
    < TI : TA | proc : KS > invkTa(TI, RI, ARGS)
 => < TI : TA | proc : none > invkTaRet(RI, RV)
if finished(KS) /\ RV := getRes(KS) /\ RI := getRequestor(ARGS) .

crl [return-smc-request]:
    < OS : TrustedOS | > invkTaRet(RI, RES) => < OS : TrustedOS | > smcRet(ARGS)
if ARGS := makeSmcArgs(RI, RES) .
```

When the RA receives the message `smcReq` with the result, it finishes the secure monitor call using the function `makeRetStmt`. The function `retVal` is used to get return values from `smcRet`.

```
crl [invoke-ta-finish]:
    < RI : RA | proc : (k($wait(F) ~> $out(XL) ~> K) KS) > smcRet(ARGS)
 => < RI : RA | proc : (k(STMT ~> K) KS) >
if RI == getRequestor(ARGS) /\ VL := retVal(ARGS) /\ STMT := makeRetStmt(VL, XL) .
```

6 A Case study on Formal Analysis of MQT-TZ

This section shows the effectiveness and feasibility of our formal model using MQT-TZ [21], a TEE-based implementation of the message transport protocol. We defined LTL properties for MQT-TZ (Section 6.1), formally analyzed them with threat models, and proposed a patch (Sections 6.2 and 6.3). Our formal specification, case study model, and experimental results are available in [25].

6.1 Overview of MQT-TZ

MQT-TZ [21] is a secure topic-based publish-subscribe protocol utilizing TEE. Figure 5 illustrates the overall architecture, presenting three entities: publisher, subscriber, and broker. Publishers collect, encrypt, and send data as messages to a broker's topic. A subscriber can receive these messages by subscribing to a topic. Brokers manage topics, subscriptions, and message delivery from publishers to subscribers. Each broker is implemented using TEE, consisting of a single RA and TA. The RA retrieves publisher messages and calls the TA for re-encryption or forward re-encrypted messages to subscribers.

The re-encryption is a key mechanism for protecting messages from potential threats. It ensures that messages cannot be exploited, allowing only the intended subscribers to read. This can be accomplished as follows: (i) Clients (publishers and subscribers) generate symmetric keys and securely share them with brokers using TLS, (ii) The publishers encrypt messages with their keys, and (iii) The brokers decrypt the messages using the publisher's keys and re-encrypt them with the subscriber's keys in TEE.

To analyze MQT-TZ, we define various requirements and express them as LTL properties. These properties are summarized in Table 1. The properties `P1` to `P5` represent requirements for correctness of message reception (`P1`, `P2`, and `P3`), system integrity (`P4`), and robustness of message sending (`P5`). `P6` is for checking whether the MQT-TZ scenarios satisfy the basic invariant.

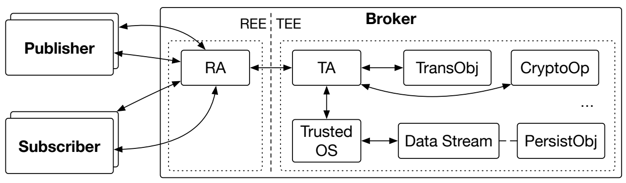

Fig. 5: Overview of MQT-TZ.

Table 1: The LTL properties for MQT-TZ.

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

For formal analysis, we represent MQT-TZ's entities (brokers, publishers, and subscribers) as Maude objects. We model brokers as instances of the `Broker` class, which is a nested object with the execution environments of Section 5 for running RA and TA, along with a buffer for storing publisher messages and a subscriber list. Publishers are modeled as instances of the `Publisher` class, which has a list of collected data to be sent to brokers. Subscribers are represented as instances of `Subscriber`, which has a list of received messages from brokers.

We specify the behavior of clients and brokers, depicted in Figure 5. For publishers, we define their behavior with two rules: collecting data, and sending it to brokers with encryption. The behavior of subscribers is represented by a single rule for message reception. We specify the behavior of a broker RA using the following rules: (1) capturing publisher messages and storing them in a message buffer, (2) running the MQT-TZ RA program, which calls a TA (explained in Section 5), and (3) receiving re-encrypted messages from the TA and sending them to subscribers.

For a broker RA and TA, we obtained their C programs from the MQT-TZ Github repository. To run them in our model, we translated a total of 1200 lines of C codes to our C-subset language using a simple translation script. Figure 6 shows the TA's re-encryption function before the conversion.

6.2 LTL Model Checking

We have performed LTL model checking for the properties in Table 1, considering two threat models. We use the following scenario for the analysis:

- Two subscribers (sub_1, sub_2), two publishers (pub_1, pub_2), and one broker participate, where the broker has two topics.
- sub_1 subscribes to a single topic, while sub_2 subscribes to all topics.
- pub_1 sends a single message, while pub_2 sends two.

```
static TEE_Result
  payload_reencryption(void *session,
                       uint32_t param_types,
                       TEE_Param params[4]){
  TEE_Result res;
  uint32_t exp_param_types =
    TEE_PARAM_TYPES(
      TEE_PARAM_TYPE_MEMREF_INPUT,
      TEE_PARAM_TYPE_MEMREF_INOUT,
      TEE_PARAM_TYPE_MEMREF_INOUT,
      TEE_PARAM_TYPE_VALUE_INPUT);

  if (param_types != exp_param_types)
    return TEE_ERROR_BAD_PARAMETERS;
  ...

  if (alloc_resources(session,
                      TA_AES_MODE_DECODE)
        != TEE_SUCCESS){
    res = TEE_ERROR_GENERIC;
    goto exit;
  }
```

```
  if (set_aes_key(session, ori_cli_key)
      != TEE_SUCCESS){
    res = TEE_ERROR_GENERIC;
    TEE_Free((void *) ori_cli_key);
    goto exit;
  }
  ...

  if (cipher_buffer(session,
  (char *) params[0].memref.buffer
  + TA_MQTTZ_CLI_ID_SZ + TA_AES_IV_SIZE,
  data_size, dec_data, &dec_data_size)
  != TEE_SUCCESS){
    res = TEE_ERROR_GENERIC;
    goto exit;
  }
  ...

  TEE_Free((void *) dec_data);
  exit:
    return res;
}
```

Fig. 6: The C code of the TA's re-encryption function.

Threat models. We consider two threat models: an out-of-memory threat and a message modification threat. The out-of-memory threat nondeterministically changes the status of a TA to `outOfMemory`. The message modification threat represents a compromised broker [21] that calls a TA with incorrect arguments. We specify the threats using Maude. For the out-of-memory threat, we model the threat as a single rewrite rule as follows.

```
rl [out-of-memory-threat]: < TK : TAKernel | status : normal >
                        => < TK : TAKernel | status : outOfMemory > .
```

For the message modification threat, we model an intruder as an instance of the `Intruder` class with a single attribute `subs-list`, denoting a broker's subscription list. Prior to the attack, the intruder learns the subscription list of a target broker from the messages in the broker's REE and records this in `subs-list`. After learning, the intruder uses this information and modifies any incoming messages of the broker by replacing the sender with any one of its subscribers. We can model this attack behavior as follows. The `modify` function replaces the `SENDER` in a publisher message `mqttzMsg` to another subscriber using the learned subscription list `SUBS-LIST`.

```
rl [message-modification-threat]: (mqttzMsg [DATA|TOPIC] from SENDER)
   < INT : Intruder | subs-list : SUBS-LIST >
=> < INT : Intruder | > modify(DATA, TOPIC, SENDER, SUBS-LIST) .
```

Model checking experiment. We consider the following threat scenarios: without any threats (`NON`), with the message modification threat (`MSG`), and with the out-of-memory threat (`OOM`). We measure the size of the state space (|`S`|) in

Table 2: The results of LTL model checking.

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	$\perp$	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	$\perp$	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

thousands, the model checking result (Safe?), and time in seconds. The $\top$ and $\perp$ denote the property is safe and violated, respectively. We use the Maude model checking command for the analysis, which provides counterexamples for violations. We run the experiment on Intel Xeon 2.8GHz with 256 GB memory.

As summarized in Table 2, the two properties P2 and P3 are violated under the threats, indicating the possible vulnerabilities. By analyzing the counterexample of the P2 violation, we have discovered that the TA can panic during the message re-encryption. This occurs because the sender of a message can be modified, leading the TA to decrypt the message with an incorrect sender's key. For the P3 violation, we have found that when insufficient memory is detected, the TA finalizes the re-encryption with an error and returns a re-encrypted message containing (dummy) data. In this case, the RA does not verify whether the TA returns a correct re-encrypted message and continues to transmit the message to subscribers, which results in obtaining the message containing dummy data.

6.3 Patching the MQT-TZ Vulnerabilities

To fix the identified vulnerabilities, we have implemented code-level patches for both the MQT-TZ RA and TA, as illustrated in Figure 7. Newly added patches are highlighted in red, while the original codes are depicted in black. The left side shows the patch for RA, and the right side is for TA. For the TA, we modify it to inform the RA of a memory error or panic. In the case of the

```
TEEC_Result
void 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);
  }
  ... }
```

```
static TEE_Result
  payload_reencryption(void *session,
                       uint32_t param_types,
                       TEE_Param params[4]){
    ...
    if (alloc_resources(session,
                        TA_AES_MODE_DECODE)
          != TEE_SUCCESS){
      res = TEE_ERROR_OUT_OF_MEMORY;
      goto exit;
    }
  ... }
```

Fig. 7: The patch codes for the MQT-TZ RA (left) and TA (right).

Table 3: The results of LTL model checking after applying the patches.

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

RA, modifications are made to ignore the re-encrypted message when a memory error or panic notification is received. Additionally, we have implemented the `discardMsg` function to handle the cleanup of the re-encrypted message.

To validate the patches, we have performed the LTL model checking from the previous section again. As shown in Table 3, `P2` and `P3` become safe (marked as red), while all other results remain the same. In addition, we observe that the state space is reduced up to approximately 185 thousand states compared to the original experiment. This is because the patches discarded the states related to memory error or panic.

In addition, we have identified redundant functions in the TA program using formal analysis. For example, `TEE_ResetOperation` is called right after allocating a cryptographic operation. Since the operation has not started, it remains in its initial state and thus the reset operation has no effect. These redundancies can be safely removed. To show this, we have collected all final states of the program with and without redundancies and compared them. We confirm the reachable states of the programs (with and without redundancies) are the same.

7 Related Work

Many studies have investigated the formal analysis of protocols leveraging TEE. The work [13] introduces a protocol for Wasm applications, and verifies the correctness of its authentication, such as aliveness and non-injective agreement. Another work [22] presents a protocol for secure remote credential management using TEE, which is verified against the Dolev-Yao model. Both papers have proven the correctness of their protocols by model checking. On the other hand, the paper [24] formally analyzes direct anonymous attestation schemes running on secure hardware through theorem proving. The papers [18,19] employ a similar approach, but aim at verifying remote attestation services of TEEs provided by Intel. However, unlike our work, they focus on specific protocols and do not propose a formal analysis framework for general TEE-based applications.

A formal analysis technique for an IoT framework using TEE is presented in [23]. It provides a hierarchical colored Petri net for Trusted IoT Architecture (TIoTA), which aims to protect data in IoT networks. This approach has been used to verify security properties in CTL by model checking. However, it is specifically tailored to TIoTA and cannot be applied to general TEE-based applications. In contrast, our work aims to provide a formal analysis framework for general TEE-based applications, written in any programming language whose operational semantics is specified in K.

8 Concluding Remarks

We have presented a formal specification for TEE APIs using Maude. We have specified two important TEE APIs (Trusted Storage API and Cryptographic Operations API) that are fundamental to mobile and IoT applications. We have leveraged Maude's object-oriented specification to reduce a representation gap between the standard document and the formal model, allowing us to effectively specify the complex architectures and behaviors of the TEE APIs.

The effectiveness and feasibility of our approach have been demonstrated through formal analysis of MQT-TZ [21,20], an open-source TEE application for IoT. We have analyzed security requirements of MQT-TZ under given threat models. Our formal analysis has revealed security vulnerabilities in the MQT-TZ implementation. We have patched a code-level bug and verified the previously violated requirements.

The future work includes providing comprehensive formal specifications for TEE APIs, covering the time API, TEE arithmetical API, and peripheral and event APIs. Additionally, we should verify the TEE API itself or generate test cases for real-world validations using our formal specification. Another important direction involves developing state space reduction techniques to enhance the efficiency of TEE application analysis.

Data-Availability Statement. The TEE formal specification, the MQT-TZ case study, and experimental results are available in [25,26].

Acknowledgments. This work was partially supported by SAMSUNG Electronics Co., Ltd., and the National Research Foundation of Korea (NRF) grant (No. 2021R1A5A1021944) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. 2022-0-00103), both funded by the Korea government (MSIT).

References

1. Ayoade, G., Karande, V., Khan, L., Hamlen, K.: Decentralized IoT data management using blockchain and trusted execution environment. In: IEEE International Conference on Information Reuse and Integration (IRI). pp. 15–22 (2018). `https://doi.org/10.1109/IRI.2018.00011`
2. Beniamini, G.: Trust issues: Exploiting TrustZone TEEs. Accessed: Aug 03, 2022 (online) (2017), `https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html`

3. Bogdanas, D., Roşu, G.: K-Java: A complete semantics of Java. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 445–456 (2015). https://doi.org/10.1145/2676726.2676982
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All about Maude - A high-performance logical framework, vol. 4350. Springer (2007). https://doi.org/10.1007/978-3-540-71999-1
5. Coppolino, L., D'Antonio, S., Formicola, V., Mazzeo, G., Romano, L.: VISE: Combining Intel SGX and homomorphic encryption for cloud industrial control systems. IEEE Transactions on Computers **70**(5), 711–724 (2021). https://doi.org/10.1109/TC.2020.2995638
6. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 533–544 (2012). https://doi.org/10.1145/2103656.2103719
7. Fitzek, A., Achleitner, F., Winter, J., Hein, D.: The ANDIX research OS — ARM TrustZone meets industrial control systems security. In: IEEE International Conference on Industrial Informatics (INDIN). pp. 88–93 (2015). https://doi.org/10.1109/INDIN.2015.7281715
8. GlobalPlatform: TEE Internal Core API Specification v1.3.1 (2021), https://globalplatform.org/specs-library/tee-internal-core-api-specification/
9. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A complete formal semantics of the Ethereum virtual machine. In: IEEE Computer Security Foundations Symposium (CSF). pp. 204–217 (2018). https://doi.org/10.1109/CSF.2018.00022
10. Hua, Z., Gu, J., Xia, Y., Chen, H., Zang, B., Guan, H.: vTZ: virtualizing ARM TrustZone. In: USENIX Conference on Security Symposium (SEC). pp. 541–556 (2017), https://dl.acm.org/doi/10.5555/3241189.3241232
11. Li, W., Xia, Y., Lu, L., Chen, H., Zang, B.: TEEv: Virtualizing trusted execution environments on mobile platforms. In: ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE). pp. 2–16 (2019). https://doi.org/10.1145/3313808.3313810
12. Machiry, A., Gustafson, E., Spensky, C., Salls, C., Stephens, N., Wang, R., Bianchi, A., Choe, Y.R., Kruegel, C., Vigna, G.: BOOMERANG: Exploiting the semantic gap in trusted execution environments. In: Network and Distributed System Security Symposium (NDSS) (2017)
13. Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: WaTZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone. In: IEEE International Conference on Distributed Computing Systems (ICDCS). pp. 1177–1189 (2022), https://doi.ieeecomputersociety.org/10.1109/ICDCS54860.2022.00116
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F
15. Nguyen, H., Ivanov, R., Phan, L.T., Sokolsky, O., Weimer, J., Lee, I.: LogSafe: Secure and scalable data logger for IoT devices. In: IEEE/ACM International Conference on Internet-of-Things Design and Implementation (IoTDI). pp. 141–152 (2018). https://doi.org/10.1109/IoTDI.2018.00023
16. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012

17. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: what it is, and what it is not. In: IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 57–64 (2015). https://doi.org/10.1109/Trustcom.2015.357
18. Sardar, M.U., Faqeh, R., Fetzer, C.: Formal foundations for Intel SGX data center attestation primitives. In: International Conference on Formal Engineering Methods (ICFEM). Lecture Notes in Computer Science, vol. 12531, pp. 268–283. Springer (2020). https://doi.org/10.1007/978-3-030-63406-3_16
19. Sardar, M.U., Musaev, S., Fetzer, C.: Demystifying attestation in Intel Trust Domain Extensions via formal verification. IEEE Access **9**, 83067–83079 (2021). https://doi.org/10.1109/ACCESS.2021.3087421
20. Segarra, C., Delgado-Gonzalo, R., Schiavoni, V.: MQT-TZ fork of the open source Mosquitto MQTT broker leveraging ARM TrustZone, https://github.com/mqttz/mqttz
21. Segarra, C., Delgado-Gonzalo, R., Schiavoni, V.: MQT-TZ: hardening IoT brokers using ARM TrustZone. In: International Symposium on Reliable Distributed Systems (SRDS). pp. 256–265 (2020). https://doi.org/10.1109/SRDS51746.2020.00033
22. Shepherd, C., Akram, R.N., Markantonakis, K.: Remote credential management with mutual attestation for trusted execution environments. In: IFIP International Conference on Information Security Theory and Practice (WISTP). Lecture Notes in Computer Science, vol. 11469, pp. 157–173. Springer (2019). https://doi.org/10.1007/978-3-030-20074-9_12
23. Valadares, D.C.G., Sobrinho, Á.A.d.C.C., Perkusich, A., Gorgonio, K.C.: Formal verification of a trusted execution environment-based architecture for IoT applications. IEEE Internet of Things Journal **8**(23), 17199–17210 (2021). https://doi.org/10.1109/JIOT.2021.3077850
24. Wesemeyer, S., Newton, C.J., Treharne, H., Chen, L., Sasse, R., Whitefield, J.: Formal analysis and implementation of a TPM 2.0-based direct anonymous attestation scheme. In: ACM Asia Conference on Computer and Communications Security (ASIA CCS). pp. 784–798 (2020). https://doi.org/10.1145/3320269.3372197
25. Yu, G., Chae, S., Bae, K., Moon, S.: The artifact of TEE formal specification (2023). https://doi.org/10.5281/zenodo.10462106
26. Yu, G., Chae, S., Bae, K., Moon, S.: Supplementary material. (2023), https://github.com/postechsv/tee-formal-spec