IET Information Security

Case Study

IET Journals
The Institution of Engineering and Technology

Build a trusted storage system on a mobile phone

ISSN 1751-8709
Received on 23rd February 2018
Revised 6th May 2018
Accepted on 5th June 2018
E-First on 16th October 2018
doi: 10.1049/iet-ifs.2018.5031
www.ietdl.org

Qiang Zhang[1] ✉, *JianZhong Qiao*[1], *QingYang Meng*[2]

[1]*School of Computer Science and Engineering, Northeastern University, ShenYang, People's Republic of China*

[2]*Company of Qianchuan Room 909, No. 155-5, Innovation Road, Shenyang District, Free Trade Zone, Liaoning Province, People's Republic of China*

✉ *E-mail: zqzq53373931@163.com*

Abstract: The authors introduce their design, implementation and formally verification of a Trusted Execution Environment (TEE)-based trusted storage system (TSS) in mobile devices, which conforms to GlobalPlatform specifications. The authors' TSS provides not only authenticating the integrity and freshness of data but also many security storage operation properties like atomicity operations of a persistent object. To improve data store efficient when a big persistent object is read or written, a new mechanism that dynamic allocate continuous memory in REE's kernel memory space and map the address to the TEE through a communication pipe is proposed. This method can reduce switching times, allocating memory times and copy memory overloads between two worlds. A formal method is used in their design and development to guarantee the correctness and security of TSS. They consider the functional correctness mainly in this study, and use traditional formal verification tool – VCC verify the functional correctness of TSS. Their evaluation demonstrates its advantage compared to existing systems in addition.

1 Introduction

In recent years, with the rapid development of the Internet of Things (IoT), great changes have taken place in smart mobile devices. To people's daily life it brought many great conveniences and is changing people's way of life. However, it also brought to hidden dangers in some fields like payments and privacy data. Trusted storage as the key function to these conveniences also facing a huge challenge. So, how to improve data's secure storage level and keeping functionality is a key challenge to facing in future years. Current mainly technologies such as cloud-based storage, virtualisation, SE (Secure Element), and TEE (Trusted Execution Environment) are used to enhance the security level of data storage. However, there are big security and functionality difference among these technologies as follows:

i. Cloud-based data storage is a feasible method for outsourcing data storage to third party providers. Yang *et al.* [1] propose a design for authenticating data storage using a small piece of high-performance trusted hardware (S-P chip model) attached to an untrusted server, but the integrity and freshness of data are guaranteed by the client user, sometimes client user is also in danger. Cloud compute [2] is based on multiple threads fast parallel access, and these threads can share resources like disk, memory and network. Cloud computing services are mainly based on the network, once the network is running unstable, then the impact on cloud computing services is also unstable.
ii. Mobile virtualisation techniques [3, 4] can provide isolation and basic resource in a single physical machine for multiple operated system (OS) or process running. So, trusted storage system (TSS) can run on the isolated domain to prohibit the attacks from other area. Park *et al.* [5] have defined a secure domain (SD) utilising domain separation technique of virtualisation, the system is divided into general domain and SD in a mobile device utilising domain separation technique of virtualisation, and SD provides a secure execution environment to protect sensitive data and secure services. The biggest defect is that there is no trusted computed base (TCB) contains the whole hypervisor exist, once the user gets the access control of the hypervisor the virtualisation method will become useless.
iii. An SE [6] is a tamper-resistant platform (typically a one chip secure micro-controller) capable of securely hosting applications and their confidential and cryptographic data (e.g. key management) in accordance with the rules and security requirements set forth by a set of well-identified trusted authorities. Some works [7–9] implement the Mobile Trusted Module, a secure element specified by the Trusted Computing Group. Such systems have the highest secure level, but they impose strong constraints on the functionalities, compute capabilities, the connectivity or the resources available within the secure area. It can store a small amount of personal data for its constraints on storage space. So, it is not versatile enough to build trusted storage module.
iv. Another effect way to create TSS solution is to rely on a trusted execution environment to encapsulate encryption and decryption services. A TrustZone-based TEE [10–13] can ensure that sensitive data is only stored, process and protected by authorised software. Using ARM TrustZone technique, the same processor is divided into two execution states, secure state and non-secure state. The secure storage system can run on the secure state separated from other system (e.g. android) run on the non-secure state we called Rich Execution Environment (REE). TEEs embedded on personal devices are ideal to support TSSs for personal data. Unlike secure crypto-processors, TEEs make it possible to (i) run complex authorised software on the TEE like an OS; (ii) access to all the mobile's device drivers from the TEE, thus enabling secure interactions between Client Applications (CAs) and Trusted Applications (TAs) running in the TEE's secure area; and (iii) control and monitor the REE from the TEE. Stated thus, according to the different technique features, TEE has obvious advantages in functionality and security. We will focus on TEE-based TSS in this paper.

Although a TEE-based architecture provides us with the guarantee of a secure area to create TSS software, However, there are some disadvantages in these secure storage systems which have been implemented currently. Sum up as follows: (i) weak functionality and low capability: It is generally accepted that a system providing trusted storage should guarantee data confidentiality, integrity, availability, and durability. However,

Table 1 Comparison of different TEEs and their respective trusted storage features. *Italic* font presents industrial TEE and normal font presents academic TEE

TEE & Owner	Trusted storage feature	In-world communication
Obc(Nokia)	Sealing storage[a] using AES-EAX authenticated encryption. The root key is derived from a one-time programmable (e-Fuse) persistent on-chip key.	Proprietary interface
<t-base(Trustonic)	Sealing storage which is not based on file systems. Instead, the unit of storage is an object. Objects are organised into a tree-like structure. Containers are protected by the secret key of their parent.	none
QSee(Qualcomm)	unknown	GP internal API and client API
OP-TEE(Linaro)	Unit of storage is block object. There is a metadata as the organise of these objects.	GP internal API and client API
TLK(Nvidia)	sealing storage	proprietary interface
open-TEE (Intel& Aalto University &University of Helsinki)	unknown	GP internal API and client API
andix OS (TU Graz University of Technology)	Merkle-Tree and AE[b]. TA uses an SBD library to implement trusted storage.	GP internal API and client API
TLR (Microsoft Research)	sealing storage	.NET remoting
safeG(Nagoya University)	unknown	secure RPC
droidVault	provide AE	proprietary interface

[a]**Sealing Storage** ensures the confidentiality, integrity and freshness of stored data are guaranteed and only authorised entities can access the data. Sealed storage is a common way to implement it. Sealed storage includes three components: (i) secret key that can never be accessed out of TEE; (ii) cryptographic mechanisms, such as authenticated encryption algorithms; and (iii) data rollback protection mechanism, such as replay-protected memory blocks (RPMB see [21]).

[b]**AE** (Authenticated Encryption) schemes achieve both data confidentiality and data authenticity simultaneously.

most of currently TSS solutions we have inferred focus only on the data confidentiality and data integrity, and with little regard for data freshness mechanism and atomicity operation of storage and so on. (ii) Lack of necessary methods of verification: it is a traditional method to use project manage, test and code review to enhance the security of system, however, these works are constrained by the low-coverage test case, and the ability of the code reviewer, it can decrease the number of bugs, but can never completely eliminate bug. Formally proving [14] is a good method to ensure the correctness of system. Unfortunately, most of the software verification needs the mathematical knowledge (e.g. math modelling and deduce reasoning etc.) of formal verification for developers and come with a non-trivial cost. In this paper, we have tried our best to solve these problems above. We have developed the TSS for ISEE2.0 [15], which is a trusted virtualisation TEE OS system based on microkernel technique and trusted compute theory. In this paper, we made the following contributions:

i. *Design and implement our TEE-based TSS system with many trusted storage features.* We describe the system architecture, data structure and implementation methods of the TSS system in detail in Section 3. This component meets the GlobalPlatform (GP) TEE System Architecture specification. It also provides the TEE Internal core API v1.1 as defined by the GP TEE standard for the development of TAs. The Trusted OS is accessible from the Rich OS(Android) using the GP TEE Client API Specification v1.0, which also is used to trigger secure execution of applications within the TEE.
ii. *We formally verified the functional correctness of TSS in Section 4.* To the best of our knowledge, this is the first TEE-based TSS system that proof system's security used of formal methods. To reduce developer effort, we use state-of-the-art tools for automated software verification, such as VCC [16], Boogie [17], and Z3 [18]. These tools need much less guidance from developers than interactive proof assistants used in previous work like Coq [19] or Isabelle/HOL [20].
iii. *To demonstrate the feasibility of our system, we built our dedicated test TA and CA.* We have written more than 5000 test cases, which include fuzzing test, abuse test and memory test. And we have done a combined test of the TEE system (include TSS) together with China Telecommunication Technology Labs (CTTL).

The remaining of this paper is organised as follows: Section 2 will introduce related works of this paper, which include ARM TrustZone-based TEE, current development status of TSS and formal methods related to our TSS. In Section 3, describe how we designed our secure storage object's data structure, storage structure, big data's read/write method and the function implementation of TSS includes atomicity of storage operation and how to ensure data confidentiality, data integrity and data freshness. In Section 4, we will describe how we verify our TSS using different tools and methods. In Section 5, we evaluated our test results. Finally, our conclusions are given in Section 6.

2 Related works

2.1 TEE-based TSS

The trusted storage modules in these TEEs have their own characteristics. In Table 1, we described industrial and academic TEE's trusted storage features and made a comparison of these projects. Because of industrial TEEs are closed source and we have little knowledge about the implemented APIs or the available hardware platforms, we focus analysed and compared TEE-based trusted storage characteristic on academic articles.

The TSS has just only been referred to Open-TEE, but without the implement method and the features of software. Current trusted storage version 2.1 [22] (see also [23] a slide-share) of OP-TEE supports two ways of file storage, there are two compile options, *CFG_REE_FS* and *CFG_RPMB_FS*, which may both enabled to store persistent object through REE file system and Replay Protected Memory Block (RPMB) separately. The RPMB way supports data rollback attack protect and the second does not support. Persistent objects are stored as many data blocks, and each data block is a file when the CFG_REE_FS is enabled. A disadvantage of this approach is that too many block files are contained in a persistent object and every block needs a switching operation to complete read or write operation. RPMB approach is a good way to keep a rollback attack, but due to the limitations of RPMB space, storing all the data in RPMB is not realistic. Andix OS provides a Secure Block Device (SBD) [24] library to TA to implement data's secure storage and use well-known techniques (e.g. Merkle-Tree & AE) to achieve authenticated storage. The idea was based on the assumption that not every TA might need a fully blown, fast random block access for its storage data. Obviously, this scheme limits the developer's behaviour of TA. DroidVault [25] provides only a trusted storage based on AE, but without guaranteeing data freshness. Our TSS system has some different parts on data structure and design from which have introduced above as follows:

- Our TSS satisfied sealing storage, which guarantees not only the confidentiality, integrity, authenticity of the data stored but also atomicity of the operations when modifying the storage.
- We use RPMB mechanism to protect data freshness, so there is no use to keep a Merkle-Tree like hash data structure in memory or disk. And a data update operation just needs to modify the RPMB counter information and data itself. Once the tree is damaged by some attacks, all of the storage data are also destroyed because of the Merkle-Tree is saved in the file system of REE.
- In our design, each persistent object corresponds to a physical file. Unlike OP-TEE, this type of storage can greatly reduce the number of switching times when reading and writing.

2.2 Formally verification

At present, TEE has become the target of attack from some hackers, some branded mobile devices have been exposed to some safety vulnerabilities [26] (see also [27, 28]) in TEE. Formal verification is recognised as the strictest method to improve software quality and ensure software safety [29]. More and more methods of formally verification and verified software's have been proposed and applied in academic research and practice business project. Our work is similar to verify a file system (a similar system API like open, read and write etc.) Pioneering work in file system direction includes FSCQ [30], COGENT [31, 32] and Flashix [33, 34], which are all successful engineering achievements. However, almost all of these jobs need a developer to construct a proof of implementation correctness using theorem tools such as Coq or Isabelle. It's difficult for people without special mathematics foundation to reasoning mathematical expressions. At the same time, these jobs require a significant time investment. The verification of FSCQ file system took Chen *et al.* about 1.5 years. Sigurbjarnarson *et al.* [35] proposed a push-button method to verify file systems via *crash refinement*, which is amenable to fully automated satisfiability modulo theories (SMT) reasoning. We use the same proofing tool Z3 [18] to simplify the difficulty of verification. However, there are also many different points between two TSS and file system (two worlds are provided to cooperate to finish the storage management of file objects, the AE and rollback protect works are in TEE and file object storage management in REE). In this paper, we consider the functional correctness mainly in this paper and use traditional formal verification tool – VCC which integrate SMT Z3 combines Coq to verify the functional correctness of TSS. VCC has successful verification works in many ways, which includes OS kernel, cryptographic protocols and application software. In [36], the author presents a framework based on VCC tools to verify the C kernel. PikeOS [37] is an embedded real-time operation system in which functional correctness is verified by VCC. In [38], the authors apply refinement methodology to prove the functional correctness of FreeRTOS by VCC. In [39], OSEK/VDK real-time operate system APIs' specification is verified by VCC from source code.

3 Implementation

3.1 How it works

The secure environment only provides limited secure storage, which is not practical to store all the sensitive data. Therefore, it's necessary to extend the trusted data storage with the help of the rich OS file system in REE. And this design method can minimise the TCB of TEE.

In Fig. 1, the rounded is the mainly function module in TSS. In the secure world, we designed GP trusted storage interface as a library. According to the architecture of MicroVisor system, the library, TA and any services that the library needs to access are all in EL0. The properties of TSS have all implemented in GP-API library. TSS interface provides RPMB access service, key service and file access service for top GP library. Due to the capability-based access mechanism of microkernel-based MicroVisor's system architecture, we need to write the access policy properly in the configuration file to make it work. In a normal world, CA invokes TA's functionality by accessing the GP client API. A *uTDaemon* that includes RPMB and VFS block device must be provided, which can help us read/write corresponding data information into each storage device and invoke the specified driver by parsing the command in command pipe. Between two worlds, a fixed size of shared memory is allocated in advance as the communication pip and data pipe. If the memory size of a data storage object is bigger than the length of the pipe's transmission, the data object memory buffer will be transported more than one time through the data pipe. We will introduce the detail working process of TSS from file storage structure, implementation of sealing storage and atomic operation in the next subsections.

3.1.1 File storage structure and format: We extend the trusted storage with the help of the Android file system. In Fig. 2, each TA has his own separate storage space as a folder, the format of folder like '/data/thh/tee/UUID'. All of the persistent object data belong to this UUID (Universally Unique Identifier of this TA) are stored in this folder as a file, the file is named with the encrypted objectID. The composition of a file object consists of two parts, the plain text partition and cipher text partition, Fig. 3 describes the detail format of the secure storage object. The real data part is consisted of attribute data partition and buffer data partition, and the length of them may longer or shorter due to the use in reality, so these two parts are stored in a way of blocks. It can improve update efficiently. We set the size of the real data block to 4Kib, and the max TEE file size is 4MB (4Kib*1024).

3.1.2 Data confidentiality: Sine persistent objects are stored in REE. The files must be stored in an encrypted form. We use AES-CTR algorithm to encrypt or decrypt our storage data to guarantee data confidentiality.

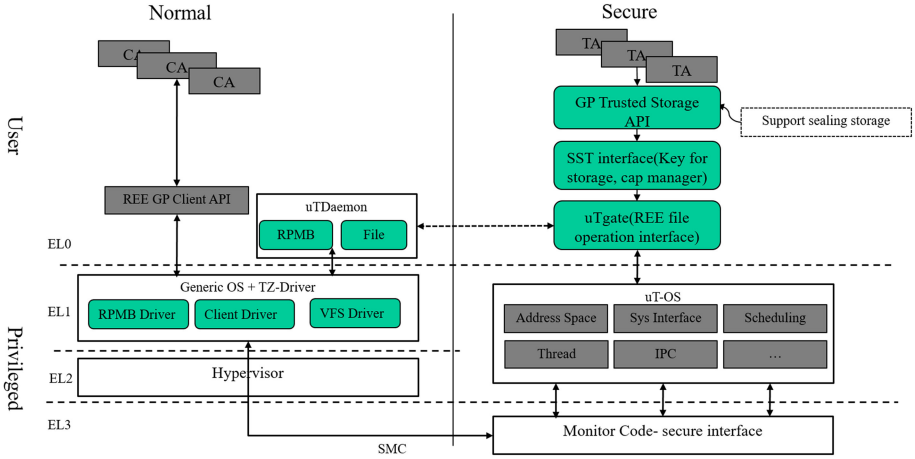

Fig. 1 *System architecture and modules of TSS*

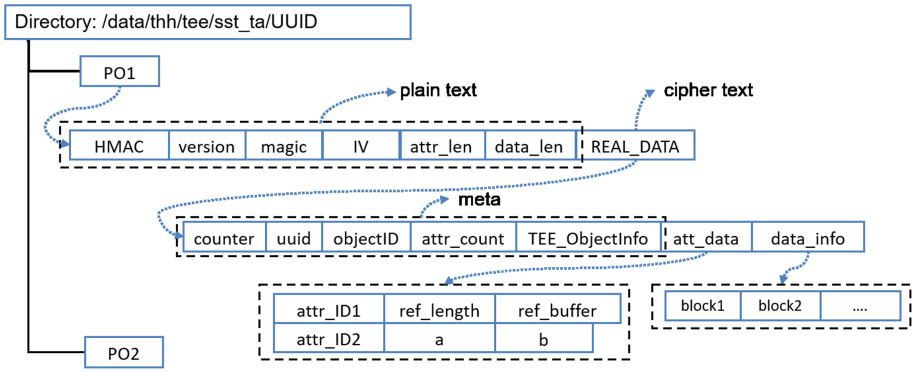

Fig. 2 *File storage architecture and file format*

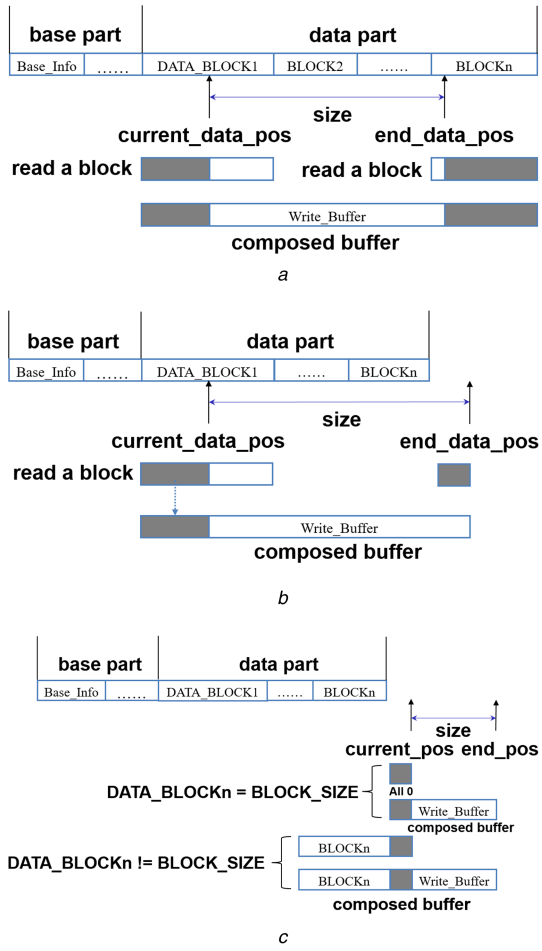

Fig. 3 *Three different kinds of data position indicator in a write operation*
(a) Data position indicator in a block and the length of write in the range of the file length, ***(b)*** Data position indicator in a block and the length of write beyond the file length, ***(c)*** Data position indicator beyond the file length

The use of trusted storage key. Each of TA must have a unique cipher key to encrypt/decrypt the storage data object which belongs to them. The TSS derives keys for each TA from the trusted storage key which is the one of eight master keys generated and stored in secure memory (the memory area used only for TEE which includes microkernel, service, TA and so on, the eight keys are stored in this memory area) and management by the key service. These eight keys are derived from the HUK (efused in the rom of the chip at the factory) using a PBKDF2 algorithm which conforms to RFC 2898 [40] at boot time. In these eight keys, some of them are reserved, some are dedicated to special functions or services. In TSS, we generate our cipher key(*fkey*) for each TA use of KDF algorithm which based on a Hash-256 algorithm. The derived algorithm is shown below, the salt is generated by Random Protected Number Generator (RPNG) and the length is 32 bits. We set the iteration count to 10,000. The length of the fkey is 32 bytes by default

$$KDF(trusted\_storage\_key, uuid||salt, /*out*/fkey) \quad (1)$$

Initialisation Vector (IV) in AES-CTR. As described above, we use AES-CTR algorithm to encrypt/decrypt our storage data. So, an IV must be generated when encrypt or decrypt a file. In our design, a 16-byte randomness number is used as IV for every persistent object, we stored it in the plain text part which mentioned in Fig. 2. This IV is length enough for use in our encryption algorithm. The initial format of a random IV is like this, *{xxxxxxxx0xxxxxxx}*. The *x* represents a random byte, so if the front 8 bytes keep unchanged, at least 2^{31} IVs will be generated in the case of using the method of add 1 to change the IV for each of blocks in a file. In order to ensure the stability of the system and without loss of availability under the premise, the real data part is limited to 4 megabytes in size, so it is enough to encrypted any persistent object file in our design.

3.1.3 Data integrity: A secure storage interface in GP API is similar to the API of the standard Unix file system. it defines the *creat*, *open*, *read*, *write* and so on. The read and write operations are basic for other operations. Due to the secure storage characteristics, the open operation includes the read process (read the entire file to verify the integrity of data in TEE) in practice. The Hash of Message Authentication Code (HMAC) method was used to guarantee the integrity of storage data. The Message Authentication Code (MAC) is calculated by using the following hash values which include the metadata, the attribute data, and the buffer data blocks as described in Fig. 2. When opening a persistent object, the first thing to do is calculate the integrity of the storage data object. The hash values calculated by these messages are all encrypted. That is, the data is first encrypted and then calculate the encrypted data hash value. The key used by the HMAC algorithm is consistent with the key used for encryption or decryption on the AES-CTR algorithm.

The integrity of file storage is checked when the persistent object opening and reading, the persistent object's block HASH values are maintained in a way that is linked as a list when opening this persistent object, in the next reading process, the hash values of blocks that read from the file of this persistent object are compared to the hash values that saved in HASH list. Beyond that, the process of reading persistent object may also exist during the writing of a persistent object due to the persistent objects are stored as blocks, as a result, a full block needs to be read. Figs. 3*a–c* show the possibility of three kinds of write operations.

3.1.4 Replay attack protected:

In order to implement the data freshness property, an RPMB is used to store the objectID and the corresponding counter value of the persistent object. RPMB is a special partition of the eMMC [41], it is used to store some sensitive data, which can be accessed only if it has access right to it. The size of RPMB space is limited, as shown in Fig. 4, we have only 544Kibs which from address *0X16F000* to the end address of RPMB for data secure storage in

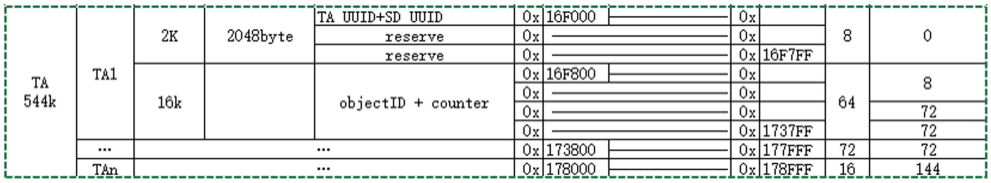

Fig. 4 *RPMB storage space layout of TSS*

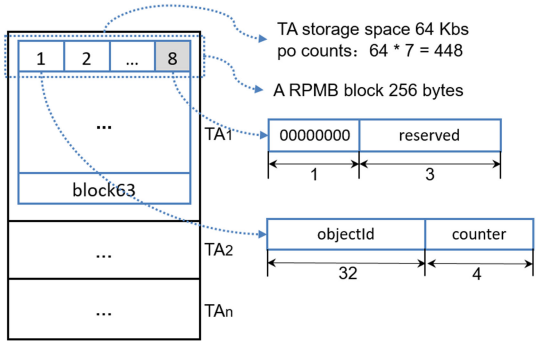

Fig. 5 *RPMB storage space layout of TA*

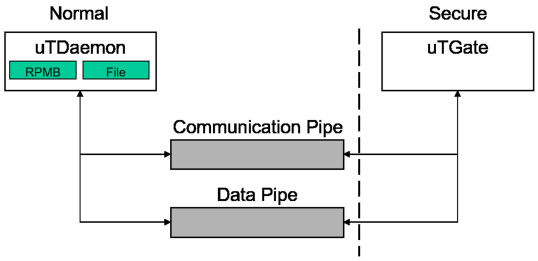

Fig. 6 *Communication and data pipe of TSS*

our design architecture. Every TA has 18Kibs to store data information which consists of 2Kibs' head partition and 16Kibs' data partition. So, it can store 30 TAs' data information at most.

The GP specification specifies that the maximum length of the objectID is 64 bytes and every TA has only 16Kibs to store the *objectID&counter* value in design. So, it can store about 240 counter data information if we use the objectID itself directly. In order to improve RPMB storage space utilisation, TSS stores the Hash of objectID instead of directly store objectID itself.

Since the minimum space for RPMB to read and write is 256 bytes as a block, we can store 7 *objectID&counter* values which the size of each is 36 bytes and the remaining 4 bytes of space is used to store meta information. Since RPMB data storage space for a TA is 16Kibs, the number of persistent objects that can be stored in a TA is 448. In Fig. 5, it describes the data structure in detail. In an RPMB block, seven persistent object counters are stored side-by-side and in the last 4 bytes of each RPMB block, we define the first byte as a tag and the remaining 3 bytes is used as an extension. In the tag, for a binary number, it defines that, from left to right, if the *i*th bit is 0, then the *i*th position does not exist persistent object counter information in this RPMB block, otherwise it is present. For instance, a decimal number 128, converted to a binary number is *10,000,000*, which means that the only first 36 bytes have existed a persistent object's *objectID&counter* value.

3.1.5 Atomicity of TSS: In the TSS system, atomic operations commonly have a succeed-or-fail definition during these APIs executing, they either successfully change the state of the system according to specification, or have no apparent effect on state space. According to the requirement of *GPD_TEE_Internal_Core_API_Specification* version 1.1, Six API must be an atomic operation. It is not simple to implement the atomic operation on these APIs, as the execution process of these APIs must consider three different parts which include operations on RPMB, memory object and file data.

Atomic operation on RPMB, memory object and file. The atomicity of trusted storage is a guarantee of when the GP-API of TSS success invoke, the state of TSS convert to the correct state, otherwise, its state consistent with the original state. The key to ensuring atomicity is how to roll back to the original state when an error occurs or power off during API execution. According to the design of TSS, the state of TSS which mentioned above consists of three update event, which are the atomic operation on an *objectID&counter* value in RPMB, memory object of TSS in TEE and file object. In TSS, RPMB access mechanism has the block operation property of atomicity, i.e. writing a block of RPMB, either the entire operation completes successfully or no write is done, this is done through the RPMB hardware mechanism. Therefore, it can be confirmed, the operation of RPMB satisfied atomicity. Next, we must make sure the rest of two parts of TSS state satisfied atomic operation. The easiest and most effective way to do this is through file copy [21] and memory object snapshot.

The sequence of update events. Different execution sequences of three state update events can result in different workload and program complexity. Among these three update events, only the RPMB operation depended on hardware and does not control by TSS in TEE. In our design, the sequence of the update event is as follows:

- According to the specification of current invoked API, copy the memory object to be modified as a snapshot (i.e. *TEE_ObjectHandle object*, RPMB block and the list of a hash of persistent Object memory information in *TEE_WriteObjectData*) and modify it. If it failed when allocating memory, the specified result (*TEE_ERROR_OUT_OF_MEMORY*) will be returned as specified by the GP API.
- Copy a file object as a duplication and update it by the preorganised memory buffer (W1–W6 in [21]), without the step of deleting the old file (W7). If this process crashed, use the recovery process (R1–R3 in [21]) roll back to the old file object.
- Update RPMB block, if successful, delete the old file object duplication and release the write lock (W7, W8 in [21]) and update the memory object with the snapshot, otherwise, delete the snapshot of memory objects and rename the old file duplication to the original name. If deleting the old file object duplication failure at the success state, just simply ignore.

3.2 Improve performance

3.2.1 Policy of big persistent object read and write: It can be seen in Fig. 6, communication between the normal world and the secure world uses the command pipe and data pipe. Two pipes are constructed by shared memory, and the size of two pipes is fixed, in our architecture, both the size of the pipe is 512Kibs. Two applications of *uTDaemon* and *uTGate* are used to resolve and deal with the command with specified format and the data in each side.

The normal file reading process is shown in Algorithm 1 (see Fig. 7).

The writing process is similar to the reading process. If the size of buffer which allocated in TSS is bigger than the size of the data pipe, the read process and switching will be repeated executed several times. In this case, the TSS needs switch secure state many times and allocates or frees memory several times according to the size of the file object. In order to improve the performance overhead in this case, we designed an improved method to enhance performance. The essence of this method is to allocate a satisfied size of the memory area in REE and read or write data into this memory space, which can reduce the number of switching times

```
current_fd ←fd
current_length ← get real read len according to fd, length and
block_size
buffer_block ← allocate a buffer of current_length
if current_length > pipe_size then
    while current_length > pipe_size, do
        current_length -= pipe_size
        buffer_block = f_read(fd, pipe_size)
        buffer_block += pipe_size
    end
else
    buffer_block = f_read(fd, current_length)
end
decrypt buffer_block
read_buffer ← get read buffer from buffer_block according to offset
```

Fig. 7 *Algorithm 1: General read algorithm*

```
current_fd ←fd
current_length ← get real read len according to fd, length and
block_size
buffer_block ← allocate a buffer of current_length
if current_length > pipe_size then
    Switch to REE
    buffer_block_REE ← allocate a buffer of current_length in
REE kernel memory space
    if buffer_block_REE is NULL
        goto ALGORITHM 1
    else
        pass the address of buffer_block_REE to TEE
        Copy buffer_block_REE to buffer_block
        Release buffer_block_REE
    end
else
    buffer_block = f_read(fd, current_length)
end
decrypt buffer_block
read_buffer ← get read buffer from buffer_block according to offset
```

Fig. 8 *Algorithm 2: Big data read algorithm*

```
An interrupt is send to TEE core
interrupt inter ← current_interrupt
bool Receive_interrupt ← true
while Receive_interrupt
    if inter.type == FIQ then
        handle(inter)
    else do
        if inter.interrupt_type == SPI then
            break
        else do
            swithtoREE()
            handle(inter)
            switchtoTEE()
        od
    od
end
```

Fig. 9 *Algorithm 3: TEE core schedule algorithm*

and copy times between two worlds. The double system in TEE induces the memory area allocated in REE must be a kernel contiguous physical memory.

There are two design policies can be used to complete above operations according to the timing of memory allocates. One chooses is to allocate memory by requesting memory at boot time. It is the only way to retrieve consecutive memory pages while bypassing the limits imposed by *__get_free_pages* on the buffer size. The second method is dynamically allocated kernel memory space in REE according to file size in TEE. The memory region allocated by both methods is contiguous in physical memory in kernel address space. For the first method, allocating memory at boot time is a 'dirty' technique, because it bypasses all memory management policies *__get_free_pages(unsigned int flags, unsigned int order)* to allocate big chunks of memory. Unfortunately, allocations of large, contiguous memory buffers are prone to failure use this method.

Compare the two methods, we choose the second method, this method is more common. If you allocate big chunks of memory failure, we only need to switch to the original workflow, it is only one more switching than the original workflow process. The process of read or write may change as follows: we take the same read process as an example shown in Algorithm 2 (see Fig. 8).

```
1  typedef struct {
2    uint32_t objectType;
3    uint32_t objectSize;
4    uint32_t maxObjectSize;
5    uint32_t objectUsage;
6    uint32_t dataSize;
7    uint32_t dataPosition;
8    uint32_t handleFlags;
9    _(ghost \bool isTransient)
10   _(ghost uint32_t flags)
11   _(invariant dataPosition <=dataSize)
12   _(invariant objectType ==
         TEE_TYPE_CORRUPTED ==>
13   objectSize == 0 && maxObjectSize ==
         0 && objectUsage == 0
14   && dataSize == 0 && dataPosition ==
         0 && handleFlags == 0)
15   _(invariant is_handleFlags_correct(
         handleFlags, flags,isTransient))
         ...
16 }TEE_ObjectInfo;
```

Fig. 10 *TEE_ObjectInfo*

3.2.2 Interrupt handling: For a general architecture of TEE, a dedicated processor core is a response to running TEE, we call it TEE core. If an interrupt signal is sent to it, TEE core must correspond the interrupt immediately. An interrupt can be divided into IRQ and FIQ, TEE core is responsible for FIQ processing. If REE OS supports full target scheduling, that is each core can handle interrupts, there is an optimised schedule method to induce TEE core performance load. It can be seen in Algorithm 3 (see Fig. 9), TEE core throws away all of SPI to any other cores. The schedule algorithm can not only decrease the interrupt numbers when creates or transports data object, but also improve the overall performance of TEE.

4 Formally verification

4.1 Specification and verification

The formal specification is used to describe the desired behaviour of the system. To verify the functional correctness, we get the formal specifications of TSS from the GP's internal API standard directly. The concrete implementation refines the abstract specification of state-machine correctly using way of step-wise refinement [42]. In our design, it begins with an abstract specification of the system functionality using ghost implementations in VCC and successively refines it to a concrete implementation in C program.

4.1.1 Formally specification of GP interface of TSS: The formal specification of GP consists of two parts, the specification of function and the specification of data structures.

Data specification. There are many data structures in TSS, including the external data types specified by GP TEE and the data types specified by ourselves. TEE_ObjectInfo shows an external data type which specified by GP. In this type, a ghost *bool* type isTransient is used to indicate whether this object is a transient object or a persistent object. In Fig. 10, using many invariants to indicate that different object types need to guarantee different object properties. Another ghost uint32 type of flags is used to determine the correctness of the handleFlags. The detail specification of *TEE_ObjectInfo* can get from TEE Internal core API v1.1 [43].

The invariant pure function *is_handleFlags_correct* can ensure the flag of any opened object is correct. The detailed content can be seen in the following data structure code (Fig. 11).

Function specification. A classic specification of GP API function like If an access right conflict was detected while opening the object, a *TEE_ERROR_ACCESS_CONFLICT* result must be returned. These specifications are the basis for building our TSS and the main partition of the entire specifications. A simple way to verify this specification is used post-condition such as _(*ensures* \ *result* == *TEE_ERROR_ACCESS_CONFLICT*⇒ !*is_flags_correct* (*flags*, \ *true*)).

```
_(def \bool is_handleFlags_correct(
    uint32_t handleFlags, uint32_t
    flags, \bool isTransient)
_(requires is_flags_correct(flags, \
    true))
{
  if(isTransient){
    return (handleFlags ==
        TEE_HANDLE_FLAG_INITIALIZED ||
         handleFlags == 0);
  }else{
    return (handleFlags == (
        TEE_HANDLE_FLAG_PERSISTENT |
        TEE_HANDLE_FLAG_INITIALIZED|
        flags));
  }
})
```

Fig. 11 *Pure function to check flags*

```
_(dynamic_owns)
struct{
  _(ghost ByteString hash_list[\
      natural])
  _(ghost unsigned counts)
}list_hash;
```

Fig. 12 *Structure of List with hash information*

```
_(ensures \forall list_node * node;
    node->prev = \old(node->prev))
_(ensures \forall list_node * node;
    node->next = \old(node->next))
_(ensures \forall list_node * node;
    node->value = \old(node->value))
```

Fig. 13 *How to ensure changed hash list equals to original hash list*

4.1.2 Specification of system properties: Abstract state machine of TSS. The state-machine specification consists of two parts: a definition of the abstract system state and a definition of the external interface as API in terms of abstract state transitions. We define the abstract TSS state that using a lot of maps which is provided by VCC and limit the rang of the map by a fix-length integer. The structures that include maps (the ghost term in VCC) may represent a double link which saves the current opened persistent object or the hash array. So, we must verify the consistency of two level specifications. The consistency verification is similar to the approach in [38]. The following code is an abstract data type of list, which have a map that mapping a natural to a byte string and integer counts indicate the length of the list (Fig. 12).

The concrete specification of the double link list and the abstract specification of the map in VCC must be consistent. We illustrate this consistency through a simple example. A structure that contains a ghost map type represents the hash value in fix position. For this, abstract data structure, the abstract state does not change after calling a GP API can represent *_(ensures list_hash = \ old(list_hash))*. Compared to a double link list, if it does not change after calling a GP API, normally, three statements as follows need to be annotated (Fig. 13).

When a GP API was invoked, we must add these post-conditions to explain that all the related system state objects do not change. Another way to express it is to use the logic function in VCC, as shown in Fig. 14. We can just add the *_(ensures state_objectHandle_changed())* as the post-conditions in the C function.

In short, we can use one logic function which includes all of the state change judgment functions (e.g. *system_state_changed*()) to indicate whether the system has changed (Fig. 15).

So, we can use a post-condition such as *_(ensures result ! = TEE_SUCCESS ⇒ !system_changed())* to ensures atomic property in the real C code. We have further developed another high-level specification, which describes the cross-cutting properties that the state-machine specification must satisfy 'every update operation to a persistent object must update counter to counter + 1'. There are some global data structure variables and properties that always hold in system (e.g. each *TEE_ObjectHandle* can only be contained

```
//storage object change logic
_(logic \bool
  state_objectHandle_changed() = (
  \forall list_node * node;
  node->prev = \old(node->prev)
  && node->next = \old(node->next)
  && node->value = \old(node->value)
))
```

Fig. 14 *Pure function to ensure objectHandle not changed*

```
//current system state space
_(logic \bool system_state_changed()=(
  //state of handle
  state_objectHandle_changed()
  //state of object hash list
  &&state_hashList_changed()
  //others system state
  && ...
))
```

Fig. 15 *Pure logic to ensure system state space not changed*

once in a list), which can be seen as the part of the system properties. We define this part of system properties in a global logic function, named it *state_hold()*. A specification of *state_hold()* like *_(pure \ bool state_hold() = property_1 && property_2 && property_3)*. The *property_x* is consisted of properties of ghost abstract global variables and properties of concrete global variables.

4.2 Modular verification of functions

A GP API consists of multiple functions, and these functions may call any other functions. The modular verification of functions method was used to verify each function separately. The following conditions must be satisfied if a function is verified to be correct: any post-condition must be satisfied when a caller verifies a function. Any objects' invariant will not be broken if the object's state has changed. The system properties still hold after executing a function. We do not need to care about the function's body statement in the verified function, but the post-conditions in any of the verified functions must be correct. The system properties of *state_hold()* must be satisfied in every function. As you saw the following code, a GP API of *TEE_CreatePersistentObject* is shown in Fig. 16, which includes three kinds of specifications that mentioned above. We do not need to be concerned with the implementation of specific content within the API, for example, for an encryption operation, we only need consider how to handle the function when the encryption fails or success.

4.3 Cryptographic library verification

Hashing. Our SHA-1256 conforms to FIPS 180-4 [3], HMAC conforms to FIPS 198-1 [44].

Aes enryption operation. Our AES256-CTR conforms to FIPS.197 AES256 [45].

The similar way of Ironclad App [46] was used to prove these cryptographic libraries. The data state machine includes all of the data information, when the data is processed, the operation of each step changes the state space, and the state-space property must meet the specification of the algorithm itself. We focus on the functional correctness of these libraries and transfer Dafny [47] code specification of a cryptographic algorithm to VCC in a way of ghost code of state machine.

4.4 Result of verification

If all of the invariants that the function inferred successfully verified and the system properties hold, we can say that it satisfies the function correctness. We have specified 17 functions which TSS related and 5 data types, the specification of 6 transient object functions is not complete. The total code line of TSS libraries is beyond 3000 lines and VCC code beyond 9000 lines. We have verified four persistent object functions which include *TEE_OpenPersistentObject*, *TEE_CreatePersistentObject*, *TEE_RenamePersistentObject* and

TEE_CloseAndDeletePersistent-Object and five generic object functions which include *TEE_Get-ObjectInfo*, *TEE_RestrictObjectUsage*, *TEE_CloseObject*, *TEE_Get-ObjectBufferAttribute* and *TEE_GetObjectValueAttribute*.

5 Evaluation

5.1 Experimental setup

We have tested our TSS on the MTK6797/Helio X20 platform, which is a platform for a business mobile. It is the industry's mobile phone chip designed with three layer architectures, in addition to the 4 + 4 core of the big/little architecture. The Helio X20 consists of two A72 CPUs with 2.5 GHz main frequency, four A53 processors with 2 GHz main frequency, and four A53 processors with 1.4 GHz main frequency for a total of 10 cores, all of which have different functionalities. The REE OS used is Android M.

To evaluate TSS's performance between different TEEs, we use trusted storage in OP-TEE as a comparison target. Because of platforms which OP-TEE supported do not contain MTK6797. QEMU is chosen as the test platform for both TEEs, and ISEE was modified by our engineers to run on the QEMU. We will evaluate our TSS both on reality hardware platform and QEMU for comparison.

To demonstrate the feasibility of TSS, we have built our dedicated test TA and CA for testing. We, not only concern on function correctness test but also the performance of the function. We test the functionality of our TSS system with two test suites, one used internally by the team of the project which we called MDTester and another for the GP-based test standard, which we have implemented it. The test cases are all implemented in TA in our MDTester.

5.2 Performance

Reality hardware platform experiment. In our first experiment, we study how the persistent object size affects the latency of TSS on the real hardware platform. For this experiment, we define a workload with a single TA executing open, create, read and write operations for different persistent object data sizes. We will test the latency of these operations with *BIG_MEM_CFG* opened and closed. We use the average time (ms) of the sampled data to represent data node in figures, the number of samples is 100 times. Traditional experiments focus on different parts of data processing, such as encryption, integrity and others. However, TEE-based security of storage extends the secure data storage with the help of REE's file system to manage file objects, we focus more on evaluating the entire API executed performance, which includes the invoke of the interface of the file system in REE, internal data processing in TEE and the switching consumption. Therefore, we evaluate the overall performance of the system rather than the performance of each operation (e.g. encryption and HMAC).

Fig. 17 shows the performance on random open, create, read and write GP API. In order to evaluate the different current offset position on a write operation performance. As you see, the time latency of create operation and write operation is almost the same, the write time latency is little more than create time latency, this is due to write operation may contain read operation before write to a file. In this situation, the compared result shown that it has a bit of influence on the latency of these operations, the main reason is shown in Figs. 3*a–c*. Due to the data block organisation format of file object, when current offset is not at the start point of a data block, it needs firstly reads the entire data block, then decrypts current data block and get the relevant data part of it to construct the entire data block, finally computes the HASH of this data block. Another reason for the decline in write performance is that the copy operation of the file object for keeping atomic property is also time to consume.

Fig. 18 shows the random read and write performance with *BIG_MEM_CFG* opened and closed separately. The performance of the read/write is similar to the normal read and write operation when the data size is less than or equal to 512Kib. However, when the data size exceeds 512Kib (i.e. the size of the shared data

```
1  TEE_Result TEE_CreatePersistentObject(
2  uint32_t storageID,
3  void *objectID, size_t objectIDLen,
4   uint32_t flags,
5   TEE_ObjectHandle attributes,
6    /*[in]*/ void *initialData,
7    size_t initialDataLen,
8    /*[out]*/ TEE_ObjectHandle * object)
9  /*1 GPTEE specification.*/
10 _(ensures \result ==
       TEE_ERROR_ITEM_NOT_FOUND ==>
       storageID !=TEE_STORAGE_PRIVATE))
11 _(ensures \result ==
       TEE_ERROR_ACCESS_CONFLICT ==> !
       is_flags_correct(flags,\true))...
12 //GPTEE specification Panic
13 _(ensures objectIDLen  >
       TEE_OBJECT_ID_MAX_LEN ==> State.
       panic)
14 _(ensures !is_correct_typeAndAttr(
       attributes) ==> State.panic)
15 /*2 other specification.*/
16 _(ensures copy_file_false ==>
       TEE_ERROR_CORRUPT_OBJECT)
17 _(ensures write_file_false ==>
       TEE_ERROR_CORRUPT_OBJECT)...
18 /*3 system specification, atomic
       property and global property.*/
19 _(requires state_hold())
20 _(ensures state_hold())
21 _(ensures \result == !TEE_SUCCESS ==>
       !system_state_changed())
```

Fig. 16 *An example of TEE_CreatePersistentObject*

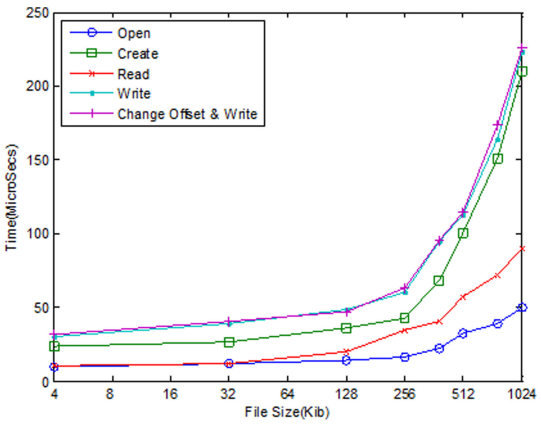

Fig. 17 *Random open, create, read and write performance*

channel), the time latency of read/write operation which using big data read and write method is significantly better than the normal read/write operation. For example, for the same file size of 4096Kib, the read/write operation shows an about 10 and 15% performance degradation separately compare to big data switch open operations, this is the most effect to the redundant world switching and memory copy operations in the normal read process. The evaluation results above are based on the actual physical platform.

In these experiments, with the increase of test data, the floating range of time also increases (from minimum 0.23 ms when 4Kib to maximum 15.753 ms when 4096Kib), This is because, during execution, there are a different number of interrupts that cause the system switch to REE handling these interrupts. The second reason is the random allocate big REE kernel memory failure cause the unstable system performance.

QEMU comparison. Current trusted storage version 2.1 of OP-TEE supports two ways of file storage, there are two compile options, *CFG_REE_FS* and *CFG_RPMB_FS*, which may both enabled to store persistent object through REE file system and RPMB separately. The first way does not support data rollback attack protect and the second support. In our experiment, we use the first way as a comparison program. Fig. 19 shows the random read and write performance between trusted storage in OP-TEE

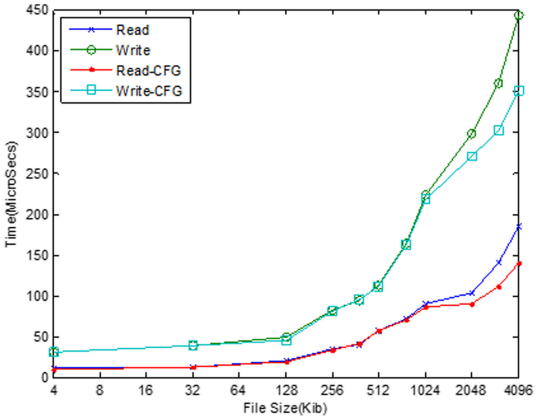

Fig. 18 *Latency of the read/write operations with BIG_MEM_CFG opened and closed*

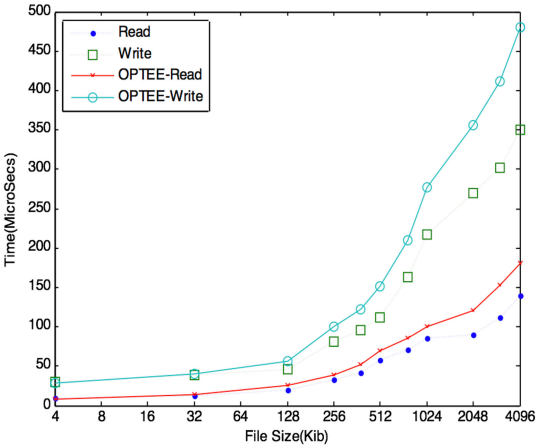

Fig. 19 *Performance comparison between OP-TEE and TSS*

and our TSS. For the same random read/write operation, the read/write performance in OP-TEE is lower than our TSS's the read and write performance.

There are two mainly reasons can describe this situation. (i) The method of big file size in our design can help us reduce memory copy times and switch times between REE and TEE; (ii) every block file needs a switching operation to complete read or write operation in OP-TEE. For example, the same 1024Kib write operation, according to the block size of 4Kib calculation, OP-TEE requires at least 256(1024Kib/4Kib) + 3(write head, attribute and synchronise operations) times of switch operations between TEE and REE. However, using our method, we only need to switch three times (two general switch (1024Kib/512Kib) and one RPMB operation switch), if we opened *BIG_MEM_CFG* and successive allocated a big continuous memory space in REE, we only need two switching times which include a write switch and an RPMB write switch. If the big memory allocated operation failure, four times of switching can complement the write operation.

6 Conclusion

At present, we have designed, implement, and verified the functional correctness of a TEE-based Security of Storage system, which based on the analysis and research of the current status of trusted storage in mobile devices. We have proposed some ways to enhance system performance by reducing switching times between TEE and REE and the memory allocated times. We have analysis our system by lots of experiments, the comparison results show the performance of our TSS is not lower than other similar system even though our design contains more security property than other similar system. In particular, compared to a similar system, formal specification and formal verification were used to prove functional correctness. In the following research work, we will complete the formal verification of the entire system, ensures the cryptographic protocol used in RPMB correctness and proving system non-interference. At the same time, further, optimise the way of file storage in REE to achieve more convenient and fast system access (e.g. a database used to save persistent object).

7 Acknowledgments

This research was supported by the Company of BeanPod, Project ISEE2.0. Thanks to every engineer and programmer who study and work to project ISEE2.0.

8 References

[1] Yang, H.J., Costan, V., Zeldovich, N., *et al.*: 'Authenticated storage using small trusted hardware'. Proc. of the 2013 ACM Workshop on Cloud Computing Security Workshop, Hangzhou, China, 2013, pp. 35–46
[2] Stefanov, E., Shi, E.: 'Multi-cloud oblivious storage'. ACM Sigsac Conf. on Computer & Communications Security, Berlin, Germany, 2013, pp. 247–258
[3] FIPS PUB 180-4, Secure hash standard (SHS), National Institute of Standards and Technology, 2012
[4] Brakensiek, J., Dröge, A., Botteck, M., *et al.*: 'Virtualization as an enabler for security in mobile devices'. Proc. of the 1st workshop on Isolation and Integration in Embedded Systems, Glasgow, Scotland UK, 2008, pp. 17–22
[5] Park, S.W., Lim, J.D., Kim, J.N.: 'A secure storage system for sensitive data protection based on mobile virtualization', *Int. J. Distrib. Sens. Netw.*, 2015, **11**, (2), p. 929380
[6] GlobalPlatform made simple guide: Secure Element. Available at http://www.global-platform.org/mediaguideSE.asp
[7] Zhang, X., Acıçmez, O., Seifert, J.P.: 'A trusted mobile phone reference architecturevia secure kernel'. Proc. of the 2007 ACM Workshop on Scalable Trusted Computing, New York, NY, USA, 2007, pp. 7–14
[8] Winter, J.: 'Trusted computing building blocks for embedded linux-based ARM trustzone platforms'. Proc. of the 3rd ACM Workshop on Scalable Trusted Computing, New York, NY, USA, 2008, pp. 21–30
[9] Dietrich, K., Winter, J.: 'Towards customizable, application specific mobile trusted modules'. Proc. of the Fifth ACM Workshop on Scalable Trusted Computing, Chicago, IL, USA, 2010, pp. 31–40
[10] Santos, N., Raj, H., Saroiu, S., *et al.*: 'Using ARM TrustZone to build a trusted language runtime for mobile applications', *ACM SIGARCH Comput. Architecture News*, 2014, **42**, (1), pp. 67–80
[11] Fitzek, A.: 'Development of an ARM TrustZone aware operating system ANDIX OS', 2014
[12] Ekberg, J.E., Kostiainen, K., Asokan, N.: 'Trusted execution environments on mobile devices'. Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security, Berlin, Germany, 2013, pp. 1497–1498
[13] Ekberg, J.E., Kostiainen, K., Asokan, N.: 'The untapped potential of trusted execution environments on mobile devices', *IEEE Secur. Priv.*, 2014, **12**, (4), pp. 29–37
[14] Woodcock, J., Larsen, P.G., Bicarregui, J., *et al.*: 'Formal methods: practice and experience', *ACM Comput. Surv. (CSUR)*, 2009, **41**, (4), p. 19
[15] ISEE, 2017. Available at http://www.beanpodtech.com/
[16] Cohen, E., Dahlweid, M., Hillebrand, M., *et al.*: 'VCC: A practical system for verifying concurrent C'. Int. Conf. on Theorem Proving in Higher Order Logics, Munich, Germany, 2009, pp. 23–42
[17] Barnett, M., Chang, B.Y.E., DeLine, R., *et al.*: 'Boogie: A modular reusable verifier for object-oriented programs'. Int. Symp. on Formal Methods for Components and Objects, Amsterdam, The Netherlands, 2005, pp. 364–387
[18] De Moura, L., Bjørner, N.: 'Z3: An efficient SMT solver'. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 2008, pp. 337–340
[19] Dowek, G., Felty, A., Herbelin, H., *et al.*: 'The COQ proof assistant user's guide: version 5.6'. INRIA, 1991
[20] Isabelle/HOL. Available at https://isabelle.in.tum.de/overview.html, accessed 2018
[21] Atomic-writes-in-a-file. Available at https://blogs.msdn.microsoft.com/adioltean/2005/12/28/how-to-do-atomic-writes-in-a-file/
[22] Forissier, J.: Optee 'Secure storage', 2016. Available at https://www.slideshare.net/linaroorg/las16504-secure-storage-updates-in-optee, accessed November 2017
[23] Optee secure storage. Available at http://connect.linaro.org/resource/las16/las16-504/, accessed November 2016
[24] Hein, D., Winter, J., Fitzek, A.: 'Secure block device–secure, flexible, and efficient data storage for ARM TrustZone systems'. 2015 IEEE Trustcom/BigDataSE/ISPA, Washington, DC, USA, 2015, vol. 1, pp. 222–229
[25] Li, X., Hu, H., Bai, G., *et al.*: 'Droidvault: A trusted data vault for android devices'. 2014 19th Int. Conf. on Engineering of Complex Computer Systems (ICECCS), Tianjin, China, 2014, pp. 29–38
[26] CVE-2015-4421, 2015. Available at http://cve.mitre.org/cgibin/cvename.cgi?name=CVE-2015-4421
[27] CVE-2014-4322, 2014. Available at http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-4322
[28] CVE-2015-4422, 2015. Available at http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4422
[29] Clarke, E.M., Wing, J.M.: 'Formal methods: state of the art and future directions', *ACM Comput. Surv. (CSUR)*, 1996, **28**, (4), pp. 626–643
[30] Chen, H., Ziegler, D., Chajed, T., *et al.*: 'Using crash hoare logic for certifying the FSCQ file system'. Proc. of the 25th ACM Symp. on Operating Systems Principles (SOSP), Monterey, CA, October 2015
[31] Amani, S., Hixon, A., Chen, Z., *et al.*: 'COGENT: verifying high-assurance file system implementations'. Proc. of the 21st Int. Conf. on Architectural

Support for Programming Languages and Operating Systems (ASPLOS), Atlanta, GA, April 2016, pp. 175–188
[32] O'Connor, L., Chen, Z., Rizkallah, C., *et al.*: 'Refinement through restraint: bringing down the cost of verification'. Proc. of the 21st ACM SIGPLAN Int. Conf. on Functional Programming (ICFP), Nara, Japan, September 2016, pp. 89–102
[33] Ernst, G., Pfahler, J., Schellhorn, G., *et al.*: 'Inside a verified flash file system: transactions & garbage collection'. Proc. of the 7th Working Conf. on Verified Software: Theories, Tools and Experiments, San Francisco, CA, July 2015
[34] Schellhorn, G., Ernst, G., Pfahler, J., *et al.*: 'Development of a verified flash file system'. Proc. of the ABZ Conf., New York, NY, USA, June 2014
[35] Sigurbjarnarson, H., Bornholt, J., Torlak, E., *et al.*: 'Push-button verification of file systems via crash refinement'. OSDI, Savannah, GA, USA, 2016, vol. 16, pp. 1–16
[36] Liang, H., Zhang, D., Jia, X., *et al.*: 'Verifying RTuinOS using VCC: from approach to practice'. 2016 17th IEEE/ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Shanghai China, 2016, pp. 373–378
[37] Baumann, C., Bormer, T.: 'Verifying the PikeOS microkernel: first results in the verisoft XT avionics project'. Doctoral Symp. on Systems Software Verification (DS SSV'09) Real Software, Real Problems, Real Solutions, New York, NY, USA, 2009, p. 20
[38] Divakaran, S.: 'A refinement-based methodology for verifying abstract data type implementations'. PhD. Thesis, Indian Institute of Science, Bangalore, 2015
[39] Shi, J., He, J., Zhu, H., *et al.*: 'ORIENTAIS: formal verified OSEK/VDX real-time operating system'. 2012 17th Int. Conf. on Engineering of Complex Computer Systems (ICECCS), Paris, France, 2012, pp. 293–301
[40] PBKDF2. Available at https://tools.ietf.org/html/rfc2898#section-5.2
[41] Emmc_partitions. Available at https://linux.codingbelief.com/zh/storage/flash_memory/emmc/emmc_partitions.html, accessed 27 November 2017
[42] Divakaran, S., D'Souza, D., Sridhar, N.: 'Efficient refinement checking in VCC'. Working Conf. on Verified Software: Theories, Tools, and Experiments, Heidelberg, Germany, 2014, pp. 21–36
[43] GPD_SPE_010, June 2016. Available at https://www.globalplatform.org/specificationsdevice.asp
[44] FIPS PUB 198-1, The keyed-hash message authentication code (HMAC), National Institute of Standards and Technology, 2012
[45] FIPS PUB 197, ADVANCED ENCRYPTION STANDARD (AES), National Institute of Standards and Technology, 2012
[46] Leino, K.R.M.: 'Dafny: An automatic program verifier for functional correctness'. Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning, Dakar, Senega, 2010, pp. 348–370
[47] Hawblitzel, C., Howell, J., Lorch, J.R., *et al.*: 'Ironclad apps: end-to-end security via automated full-system verification'. OSDI, Broomfield, CO USA, 2014, vol. 14, pp. 165–181