Library Implementation and Performance Analysis of GlobalPlatform TEE Internal API for Intel SGX and RISC-V Keystone

Kuniyasu Suzaki*†, Kenta Nakajima*, Tsukasa Oi*, Akira Tsukamoto†
*Technology Research Association of Secure IoT Edge application based on RISC-V Open architecture (TRASIO)
†National Institute of Advanced Industrial Science and Technology (AIST)

***Abstract*—Trusted Execution Environment (TEE) becomes a popular security extension on current CPUs (e.g., Arm TrustZone, Intel SGX, and RISC-V Keystone), but each TEE has its original SDK and cannot keep software portability. GlobalPlatform (GP) defines the general APIs named "TEE Internal APIs", and smartphones mainly use them. In addition, the implementation of GP API assumes a Trusted OS, and some TEEs cannot implement it directly. Furthermore, some TEEs offer Enclave Definition Language (EDL) for secure communication between a normal application and a trusted application, which is not assumed by GP APIs.**

In order to solve these problems, we propose a library implementation of GP TEE internal APIs on each TEE SDK. We selected GP APIs for architecture-dependent or independent (e.g., secure storage and time). The architecture-independent APIs require the help of Linux or can be implemented efficiently with CPU-specific instruction. They are implemented as same as possible on each architecture. The library is designed to fit for each EDL (i.e., "edger8r" on Intel SGX "keyedge" on RISC-V) and keeps the communication security. The library is implemented on Intel SGX and RISC-V Keystone. The performances are measured and compared with the OP-TEE which is a trusted OS style implementation on Arm TrustZone. The comparison shows the feature of each implementation.

***Index Terms*—Trusted Execution Environment (TEE), Intel SGX, ARM TrustZone, RISC-V Keystone, GlobalPlatform TEE Internal API, Performance Evaluation**

I. Introduction

Current OSes must support many devices and rich applications, and they become huge and complex. The OSes cannot escape from vulnerabilities because system software vulnerabilities depend on code size and complexity [1], [2], [10], [25]. Therefore, current CPUs offer a special execution environment isolated from the OS, which is named as `Trusted Execution Environment (TEE)` (Notes, some architecture calls the environment as `secure world` or `enclave`, and this paper uses the term properly in the context). The TEE runs a trusted application (TA) as a critical process on it.

TEE is included in popular commercial CPUs (e.g., Arm TrustZone [6], [22] and Intel SGX [8], [9]) as well as an open-source instruction set architecture CPU "RISC-V" (e.g., Keystone [18], [19] and Sanctum [7]). Unfortunately, each TEE has its original Software Development Kit (SDK) for a TA, and programmers must learn its Application Programming Interface (API).

On the other hand, GlobalPlatform [11], a non-profit industry association for computer security, defines some specifications for TEE, which are independent of CPU architecture. The specifications include TEE Internal APIs [12] which are popular on smartphones [14], [24]. The paper [14] reports that 60% of Android smartphone uses GlobalPlatform TEE Internal APIs. However, the design assumes a trusted OS on TEE, and some TEE architectures do not fit for it. For example, Intel SGX allows user privilege (i.e., ring3) only and cannot run a trusted OS kernel.

In addition, some TEE original SDKs assume Enclave Definition Language (EDL) which generates a glue code for secure communication between a normal application and a trusted application [4]. The EDL verifies the region of the pointer and the size of the buffer. The GlobalPlatform TEE Internal APIs do not assume the EDL because it is designed for API on Trusted OS.

This paper proposes a library implementation of GlobalPlatform TEE Internal APIs on Intel SGX and RISC-V Keystone. The APIs are selected and integrated to the TEE original SDK which includes EDL. The implementation keeps TA's interoperability and makes it possible to run on any TEE.

The performance of each API is measured on a real hardware (Intel NUC board and SiFive Unleashed board). The results are compared with OP-TEE which offers the GlobalPlatform TEE Internal APIs as a OS kernel running on Arm TrustZone of Raspberry Pi3+.

Contributions and Challenges:

1) **Separate GlobalPlatform TEE Internal API into Architecture-Independent and Dependent:** The separation enables implementing architecture-independent APIs as a portable library. The architecture-dependent APIs require the help of the host OS or can be implemented efficiently with architecture-dependent instructions. They are designed to be kept as small modifications as possible.
2) **Implement as a Portable Library:** Selected GlobalPlatform APIs are implemented as a portable library for Intel SGX and RISC-V Keystone. The implementation is based on each SDK and shows some difficulties on each architecture (e.g., EDL and time instruction).
3) **Measurement of each API on different architectures:** The performance of each API is measured on Intel SGX and RISC-V Keystone. The performance is also measured on OP-TEE which is a trusted OS on Arm TrustZone. The results are compared to show the dif-

ference of implementations and architectures. We also measured the effect of EDL.

The remainder of this paper is organized as follows. Section II describes the background knowledge. Section III depicts the selected GlobalPlatform Internal APIs. Section IV mentions the implementation, and Section V shows the performance measurements. Section VI describes security analysis, and Section VII concludes this paper.

II. Background

The brief explanations for TEE architecture, TA's API, communication method between normal and trusted applications, performance measurement in TEE are mentioned.

A. TEE Architecture

Three TEE Architectures used in this paper are described.

Intel SGX: Software Guard Extensions [8], [9] offers the single address model of TEE, which is named `enclave`. SGX allows a TA as a part of a normal application on host OS, and the TA runs on user-level (ring 3) in an enclave. Therefore, an enclave does not include a trusted OS. The TA is implemented as a library by SDK offered by Intel. The total memory for enclaves is defined by UEFI and reserved at boot time. The maximum size is 128MB (96MB for enclaves). The memory region is encrypted with the key generated at boot time.

Arm TrustZone: Since smartphones, game machines, and set-top boxes use Arm Cortex-A TrustZone [6], [22] [1] for their critical processing, it is the most popular TEE. TrustZone offers 2 world view model, i.e., secure world (i.e., TEE) and normal world (i.e., REE: Rich Execution Environment). Each world has user and supervisor mode and runs a kernel (i.e., trusted OS or normal OS). Many trusted OSes offer the APIs defined by GlobalPlatform, which are limited for trusted application (TA) programming. Some APIs(e.g., secure storage) require the help of normal OS. The memory for a TEE is allocated at the boot time in general, and the size is small because of keeping a small Trusted Computing Base (TCB).

RISC-V Keystone: RISC-V is an open Instruction Set Architecture (ISA) and has some implementations of TEE (e.g., Sanctum [7], MI6 [5], Keystone [18], [19], MultiZone [15], TIMBER-V [28]). This paper picks up Keystone because it is open source and runs on a real CPU (SiFive's Unleashed board). Keystone is a mixed architecture of Arm TrustZone and Intel SGX. Keystone can create some TEEs dynamically as SGX, although TrustZone offers only one TEE. The TEE is named `enclave` as SGX, but each TEE has user and supervisor mode as TrustZone. Each TEE has its own `runtime`, which works as an OS kernel. The memory for an enclave is dispatched from REE (i.e., Linux OS) dynamically using the Physical Memory Protection (PMP) mechanism. The dispatched memory is sanitized and used for an enclave.

[1]Cortex-M also has a different TrustZone. This paper treats Cortex-A only.

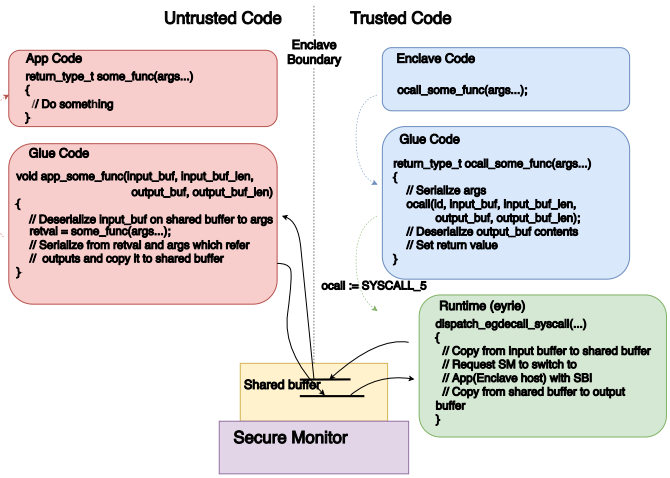

Fig. 1. Keystone OCALL

B. Trusted Application's API

Each TEE and software vendor has its own SDK for the programming of trusted applications. Intel gives the original API with its SGX SDK [16]. Microsoft's Open Enclave SDK [21] defines its own TEE API which is independent of the base TEEs because it is designed to support Intel SGX and Arm TrustZone. Google Asylo [13] has another portable TEE API on Intel SGX and AMD SEV. RISC-V Keystone also offers its own API [17].

GlobalPlatform TEE Internal API specification [12] is CPU architecture independent and well-known API because it is used on smartphones [14], [24]. The specification includes about 130 APIs with many arguments. The specification is designed to write a critical process and includes many cryptographic APIs. The GlobalPlatform TEE Internal API just defines the name of cryptographic function and does not fix the cipher suite. The cipher suite is defined by arguments. In addition, GlobalPlatform TEE Internal API has its original coding style. For example, the cipher suite and region for a key must be registered before use. The detail of coding is explained in Section III-A.

C. Secure Communication between Normal and Trusted Applications

There are several forms of communication between normal and trusted applications. OCALL, which means "out call", is a function call interface that a TA calls some normal world functions. ECALL, which means "enclave call", is the inverse direction, which is implemented on SGX but not yet on Keystone.

Figure 1 shows how OCALL works in Keystone with Enclave Definition Language (EDL). There are two glue codes both in the application and the enclave, which are created by EDL. Each glue code wraps the edge/boundary system call and serializes/deserializes the arguments/results. These

TABLE I
SELECTED GLOBALPLATFORM TEE INTERNAL API

Category	Architecture (In)Dependent	API name
Random Number	Dependent	TEE_GenerateRandom
Time	Dependent	TEE_GetREETime, TEE_GetSystemTime
Secure Storage	Dependent	TEE_CreatePersistentObject, TEE_OpenPersistentObject, TEE_ReadObjectData, TEE_WriteObjectData, TEE_CloseObject
Transient Object	Dependent Independent	TEE_GenerateKey, TEE_AllocateTransientObject, TEE_FreeTransientObject, TEE_InitRefAttribute, TEE_InitValueAttribute, TEE_SetOperationKey
Crypto Common	Independent	TEE_AllocateOperation, TEE_FreeOperation
Authenticated Encryption	Independent	TEE_AEInit, TEE_AEUpdateAAD, TEE_AEUpdate, TEE_AEEncryptFinal, TEE_AEDecryptFinal
Symmetric Cipher	Independent	TEE_CipherInit, TEE_CipherUpdate, TEE_CipherDoFinal
Asymmetric Cipher	Independent	TEE_AsymmetricSignDigest, TEE_AsymmetricVerifyDigest
Message Digest	Independent	TEE_DigestUpdate, TEE_DigestDoFinal

wrapper codes should be crafted very carefully because they are put at the attack surfaces.

These wrapper codes have rather fixed patterns and usually are generated with the tool called **edger**. Intel SGX SDK has **edger8r**, and Open Enclave uses **oeedger8r** which is a modified version of edger8r. Recently, Keystone adds **keyedge** as its edger. All edgers generate glue codes from the annotated function.

Keyedge uses **LLVM infrastructure** and **flatcc**. The former is to analyze the syntax. The latter is to generate serializing/deserializing code. On the other hand, edger8r/oeedger8r is written by **OCaml** and has its own analyzer and code generator. Edger8r and oeedger8r are relatively easy to port for the other systems. We had ported oeedger8r to Keystone and used it temporally until keyedge is released. All edgers generate wrapper codes that sanitize the arguments and the results, though there may be yet unknown issues as paper [26] shows.

D. Performance Measurement in TEE

Linux **perf** is a useful tool to measure the performance of each function but cannot work for a trusted application because trusted OS has unique system calls. SGX-Perf [27] is Intel SGX specific solution and cannot be applied to other architectures. SGX-Perf requires OCALL for logging the performance. Unfortunately, OCALL causes heavy overhead, and the performance is unstable. It is not suitable for precise performance measuring. TEE-Perf [4] is a general perf implementation but assumes a recorder process which occupies a core along with the perf process. The recorder process has a software counter on a shared memory, which is incremented by the infinity loop. The perf process gets the number of software counter via shared memory. It can be applied to REE, but the implementation is redundant. Therefore, we need the original perf mechanism to analyze a trusted application in each TEE (i.e., Intel SGX, RISC-V Keystone, and Arm TrustZone).

III. SELECTED GLOBALPLATFORM TEE INTERNAL APIS

Table I shows the list of selected GlobalPlatform TEE Internal APIs. The specification [12] has 29 categories, and we select 9 categories to implement a minimum TA.

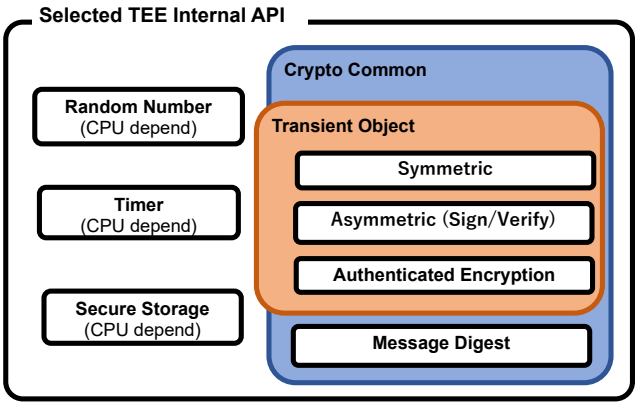

Fig. 2. Relation of Selected GlobalPlatform TEE Internal API categories

The first three categories (random number, time, and secure storage) and 1 API (`TEE_GenerateKey`) from the transient object category are architecture-dependent APIs because they require the help of the host OS or can be implemented efficiently with special instructions. They should be implemented for each architecture. The other APIs (transient object, crypto common, authenticated encryption, symmetric cipher, asymmetric cipher, and message digest) are related to cryptographic operations, which can be implemented as a portable library.

A. Dependency of categories

Figure 2 shows the dependency of selected 9 categories, and Figure 3 shows the sample program that indicates the relation of categories. The random number, time, and secure storage, which are architecture-dependent APIs, can be used standalone. However, other cryptographic APIs have dependency.

The crypto common category [2] includes the algorithm handling APIs. Therefore, generic operation is used before and

[2]The specification [12] names this category "generic operation", but we think "crypto common" is suitable in this paper.

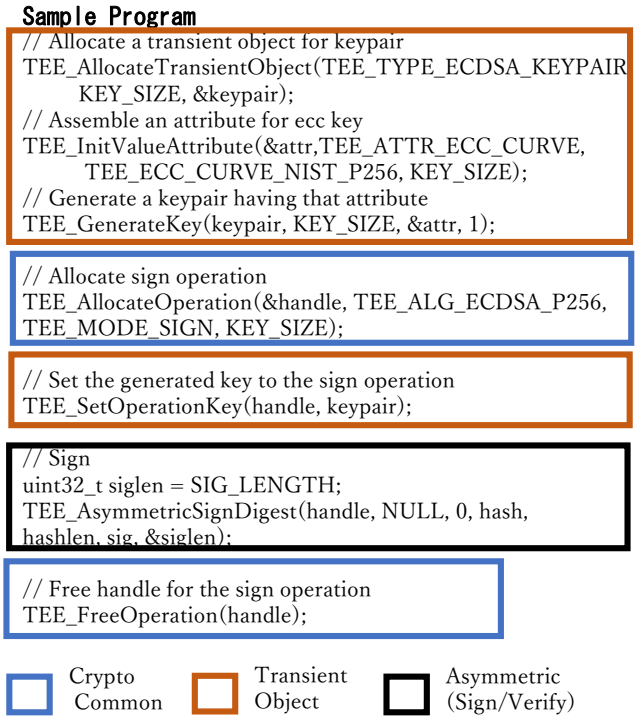

Fig. 3. Sample program that indicates the relation of categories

after authenticated encryption, symmetric cipher, asymmetric cipher, and message digest APIs. The transient object category offers APIs to handle a secret key. Therefore, authenticated encryption, symmetric cipher, and asymmetric cipher need to use transient object APIs to keep a key, but message digest does not use these APIs because it does not need to keep a key. Crypto common and transient objects are independent each other. The detail of each category is described in the following subsections.

B. Architecture-Dependent API

Architecture-dependent APIs are categorized as random number, time, secure storage, and transient object (only `TEE_GenerateKey` API).

Random Number: The random number category includes only 1 API; `TEE_GenerateRandom` which generates random data. The hardware resource for random number generator depends on each CPU architecture. In addition, some CPUs do not allow to use the hardware resource from TEE. The `TEE_GenerateRandom` must be implemented as architecture-dependent API.

Time: The time category includes 2 APIs; `TEE_GetREETime` and `TEE_GetSystemTime`. The `TEE_GetREETime` retrieves the current REE system time. The REE system time is offered by the normal OS and requires communication with the normal OS. On the other hand, `TEE_GetSystemTime` is defined to retrieve the current system time. The system time is allowed to be implemented arbitrarily. The minimum guarantee is that the system time must be monotonic for a given TA instance. The implementation is allowed to use the REE time as `TEE_GetREETime`.

Secure Storage: The secure storage category includes 5 functions; `TEE_CreatePersistentObject`, `TEE_OpenPersistentObject`, `TEE_ReadObjectData`, `TEE_WriteObjectData`, and `TEE_CloseObject`. Most TEEs have no own storage and depend on the file system of normal OS. The communication method from TEE to the normal OS depends on each CPU architecture.

The `TEE_CreatePersistentObject` creates a persistent object with initial attributes and an initial data stream content. The `TEE_OpenPersistentObject` opens a handle on an existing persistent object. It returns a handle that can be used to access the object's attributes and data stream. The `TEE_ReadObjectData` reads the data stream from an opened object to the buffer. The `TEE_WriteObjectData` writes the data stream from the buffer to an opened object. The `TEE_CloseObject` closes an opened object handle. The object can be persistent or transient.

Transient Object: `TEE_GenerateKey` in the transient object category is architecture-dependent API because it requires a random number. The random number is generated in the same way as `TEE_GenerateRandom`. The `TEE_GenerateKey` generates a random key or a key-pair and populates a transient key object with the generated key material.

C. Architecture-Independent API

Architecture-independent APIs are categorized as crypto common, transient object (except `TEE_GenerateKey` API), authenticated encryption, symmetric cipher, asymmetric cipher, and message digest. They are cryptographic APIs and can be implemented as a portable library.

Crypto Common: The crypto common is a special category on GlobalPlatform TEE Internal APIs, which includes `TEE_AllocateOperation` and `TEE_FreeOperation`. The `TEE_AllocateOperation` allocates a handle for a new cryptographic operation and sets the mode and algorithm type. The `TEE_FreeOperation` deallocates all resources allocated by `TEE_AllocateOperation`.

Transient Object: The transient object category except `TEE_GenerateKey` is architecture-independent API. It includes `TEE_AllocateTransientObject`, `TEE_FreeTransientObject`, `TEE_InitRefAttribute`, `TEE_InitValueAttribute`, and `TEE_SetOperationKey`. The `TEE_AllocateTransientObject` allocates an uninitialized transient object, i.e., a container for attributes. Transient objects are used to hold a cryptographic object (key or key-pair). The `TEE_FreeTransientObject` deallocates a transient object previously allocated with `TEE_AllocateTransientObject`. After this API has been called, the object handle is no longer valid. The `TEE_InitRefAttribute` and `TEE_InitValueAttribute` helper APIs can be used to

populate a single attribute either with reference to a buffer or with integer values. The TEE_SetOperationKey programs the key of operation; that is, it associates an operation with a key.

Authentication Encryption (AE): The authentication encryption category includes TEE_AEInit, TEE_AEUpdateAAD, TEE_AEUpdate, TEE_AEEncryptFinal, and TEE_AEDecryptFinal. The TEE_AEInit initializes an AE operation. The operation must be in the initial state and remains in the initial state afterwards. The TEE_AEUpdateAAD feeds a new chunk of Additional Authentication Data (AAD) to the AE operation. Subsequent calls to this API are possible. The TEE_AEUpdate accumulates data for an AE operation. The TEE_AEEncryptFinal processes data that has not been processed by previous calls to TEE_AEUpdate. It completes the AE operation. The TEE_AEDecryptFinal processes data that has not been processed by previous calls to TEE_AEUpdate. It also completes the AE operation.

Symmetric Cipher: The symmetric cipher category includes TEE_CipherInit, TEE_CipherUpdate, and TEE_CipherDoFinal. The TEE_CipherInit initializes the symmetric cipher operation with a key. The TEE_CipherUpdate encrypts or decrypts input data. The TEE_CipherDoFinal finalizes the cipher operation.

Asymmetric Cipher: The asymmetric cipher category includes TEE_AsymmetricSignDigest and TEE_AsymmetricVerifyDigest. The TEE_ AsymmetricSignDigest signs a message digest within an asymmetric operation. The TEE_ AsymmetricVerifyDigest verifies a message digest signature within an asymmetric operation.

Message Digest: The message digest category includes TEE_DigestUpdate and TEE_DigestDoFinal. The TEE_DigestUpdate accumulates message data for hashing. The TEE_DigestDoFinal finalizes the message digest operation and produces the message hash. Afterward, the message digest operation is reset to the initial state and can be reused.

IV. Implementation

The selected APIs are implemented on RISC-V Keystone and Intel SGX. Architecture-dependent APIs are implemented for each architecture, and architecture-independent APIs are implemented as a portable library.

A. Architecture-Dependent API

Random Number: Random number generation is an important factor for crypto algorithms, and GlobalPlatform TEE Internal API offers TEE_GenerateRandom. RISC-V has no instruction for random number, but Intel CPU has RDRAND instruction.

On Keystone, the implementation of TEE_GenerateRandom requires the help of the host OS, which causes OCALL. On SGX, the implementation uses sgx_read_rand offered by Intel SGX SDK. The behavior depends on SGX. If the CPU has SGXv2, it can use RDRAND instruction. However, if the CPU has SGXv1, it needs the help of the host OS.

Time: As time, two APIs are selected; TEE_GetREETime and Get_SystemTime. Both of them show the system clock but differ in the clock resource. TEE_GetREETime is defined to show the system clock used on REE (i.e., Linux). On the other hand, Get_SystemTime depends on implementation. Some implementations use the system clock used on REE, which is the same as Get_REETime, but this implementation has no meaning. In our design, Get_SystemTime is changed to indicate the hardware timestamp counter. The change enables to compare the overhead to get the system clock on REE and the hardware timestamp counter in TEE.

Secure Storage: As secure storage, object manipulation APIs are selected (e.g.,TEE_ReadObjectData and TEE_WriteObjectData). The secure storage is designed to use the Linux file system, which follows the OP-TEE implementation. The secure storage on SGX and Keystone requires OCALL. The security of OCALL is protected by the glue code created by edger8r on SGX and keyedge on Keystone. The glue code is created for each data and may cause overhead.

We use encrypted normal files to emulate the secure storage for SGX and Keystone because we couldn't assume such hardware for them. The file contents are encrypted with AES CBC. The file I/O operations are done with OCALL. The AES key should be persistent and unpredictable.

This scheme gives some restrictions and issues on API.

- No random access is allowed.
- Read or write is permitted only when the data size is a multiple of 16.
- Open with RW mode isn't supported. Storage (persistent object) should be opened with write-only mode or read-only mode.
- The trusted application shouldn't depend on the existence of the object. The file object can be removed by some attack or failure.

The key and the initial vector (iv) of AES cause other implementation issues. The ideal key and initial vector are hard to get in the usual Keystone environment. [3] We utilize an attestation report.

The Keystone attestation report is returned from the secure monitor (SM) when the enclave requests. It's structured as the below.

enclave_hash(64)
data_len(8)
data(1024)
signature(64)
sm_hash(64)
sm_public_key(32)
sm_signature(64)
device_public_key(32)

[3]SGX has sgx_get_key which is essentially a wrapper of EGETKEY/EREPORT instruction and uses it for the protected_fs library in SDK.

The `data_len` and `data` are given by the enclave, and the SM fills the other fields. The signature is computed with taking account of the data part. Thus the 64-byte `signature` in Keystone's report structure is an enclave/system invariant which depends on the given data. With using objectID (file name) as that data, it returns an enclave/system/objectID invariant. We deduce the key and the initial vector from this invariant. This style adds other constraints on persistent objects.

- An object can be accessed from only one enclave.
- Changes in the system could make all persistent objects obsolete.

It means that changes of SM and bootloader will give a different signature even for the same enclave.

Rollback attack is another issue. There is no automatic mechanism to avoid rollback attack with our current implementation. The user must update ObjectID or some data in the enclave explicitly for the new contents. This style will mitigate the rollback attack because key/iv are ObjectID/enclave invariants.

B. Architecture-Independent API

The implementation of architecture-independent API is based on the SDK offered by each architecture. Intel SGX SDK offers rich APIs, but RISC-V keystone SDK offers minimum APIs.

The most important factor is the cipher suite. The choice of cipher suite is based on the ongoing arguments of RISC-V cryptographic extension. They give minimal and typical interfaces that are common for SGX and Keystone. Supported cryptographic operations are 256bit AES CBC/GCM, SHA3 digest and ed25519 sign/verify only. OP-TEE gives these functions with their API. For Keystone and SGX, we use existing cryptographic libraries, though SGX prepares some of these functions in SDK. The existing cryptographic libraries are `wolfcrypt` [29] and `mbedtls` [3], and the user can select one of them at the build time.

V. Performance measurement

The performance of Selected API is measured with our original tool.

A. Perf Mechanism on TEE

We develop the perf tool to evaluate the performance of GlobalPlatform TEE Internal APIs. The perf uses the instruction to get the CPU timestamp counter. Intel, RISC-V, and Arm offer `rdtsc`, `rdcycle`, and `cntvct_el0` instruction, respectively. The clock cycle of timestamp counter is different from CPU speed because CPU speed can be controlled by setting. The timestamp counter does not change the clock cycle and keeps absolute results.

TABLE II
Target Machines (The core of Intel includes hypertheading.)

Architecture	Core	Mem	REE	TEE	Timestamp CPU Speed
Pentium J5005	4	8GB	Linux 5.3.0	SGXv2 (SDK v2.8)	1,500MHz 1,000MHz
Unleashed U540	4	8GB	Linux 4.15.0	Keystone v0.3 (with Eyrie OS)	1,000MHz 1,00MHz
Cortex-A53	4	1GB	Linux 4.14.56	TrustZone (OP-TEE 3.8.0)	19.2Mhz 1,400MHz

The evaluation is achieved on real machines; Intel NUC, SiFive Unleashed board, and Raspberry Pi3 Model B+ board [23]. Table II shows the target machine conditions for the evaluation. The table shows the setting of the clock cycle of timestamp counter and CPU speed. Keystone and SGX run on the same CPU speed (1,000MHz), but TrustZone runs 1.4 times faster (1,400MHz).

B. Performance of each API

Table III shows each API performance cycle count of timestamp and the time calculated from the cycle count on Keystone, SGX, and TrustZone. This table does not include secure storage categories because these APIs are not stable. The results of secure storage show in another table (Table IV). The cipher suite on Keystone and SGX uses `mbedtls`, and OP-TEE on TrustZone uses OP-TEE's default. This table shows precise figures, but they are varied and not easy to understand. Therefore, two figures (Figure 4 and 5) are created to show these features.

Figure 4 shows the radar chart of API performance, which is time-ordered by the RISC-V performance (from `TEE_GetSystemTime` to `TEE_AEEncryptFinal`). The radar chart does not include the symmetric and asymmetric cipher related APIs which includes `TEE_GenerateKey` because they take much time and are not easy to compare, and indicated by another figure. The radar chart indicates that most APIs on Keystone is slower than SGX and TrustZone. Some APIs from `TEE_GetSystemTime` to `TEE_AEInit`) show that TrustZone is slower than others, but the absolute times are short and do not cause a problem. The performance of SGX is comparative to the OP-TEE on TrustZone, even if the TrustZone runs 1.4 times faster. This result shows that our library implementation is reasonable.

Figure 5 shows the performance of symmetric and asymmetric cipher related APIs which includes `TEE_GenerateKey`. Since these APIs take much time and the performances are varied, the time scale is log10. The results show that asymmetric cipher APIs (`TEE_GenerateKey(ECDSA)`, `AsymmetricSignDigest`, and `TEE_AsymmetricVerifyDigest`) on TrustZone are 30-200 times slower than Keystone and SGX, even if CPU speed on TrustZone is 1.4 times faster. The results indicate the difference in software implementation. We guess OP-TEE has many chances of performance tuning.

Table IV shows the performance of APIs of secure storage categories. The performances are not stable, and the table shows the average, minimum, maximum, and standard deviation. `TEE_WriteObjectData` and `TEE_ReadObjectData` are measured with `TEE_CloseObject` to show real performance. The results show that read operations are stable than write

TABLE III
CYCLE COUNT OF TIMESTAMP AND TIME FOR SELECTED APIS ON RISC-C KEYSTONE, ARM TRUSTZONE AND INTEL SGX

Category	Architecture dependency	API	Keystone		SGX		TrustZone	
			cycle	µ-sec	cycle	µ-sec	cycle	µ-sec
Random Number	dependent	TEE_GenerateRandom	156,645	156.65	7,583	5.06	691	36.00
Time	dependent	TEE_GetREETime	73,101	73.10	39,155	26.10	317	16.52
		TEE_GetSystemTime	274	0.27	267	0.18	44	2.30
Transient Object	dependent	TEE_GenerateKey(AES)	2,262,152	2,262.15	1,261,305	840.87	4,147	215.99
		TEE_GenerateKey(ECDSA)	2,199,125	2,199.13	1,182,361	788.24	4,237,436	220,699.77
	independent	TEE_AllocateTransientObject	1,372	1.37	4,674	3.12	98	5.12
		TEE_FreeTransientObject	1,373	1.37	4,621	3.08	83	4.32
		TEE_InitValueAttribute	291	0.29	222	0.15	5	0.27
		TEE_SetOperationKey	1,206	1.21	878	0.59	330	17.18
Crypto Common	independent	TEE_AllocateOperation	1,500	1.50	2,875	1.92	376	19.56
		TEE_FreeOperation	428	0.43	587	0.39	106	5.50
Authenticated Encryption	independent	TEE_AEInit	21,163	21.16	30,622	20.41	308	16.02
		TEE_AEEncryptFinal	170,939	170.94	83,892	55.93	1,140	59.36
		TEE_AEDecryptFinal	167,353	167.35	78,387	52.26	1,054	54.90
Symmetric Cipher	independent	TEE_CipherInit	5,088	5.09	5,692	3.79	184	9.60
		TEE_CipherUpdate	105,788	105.79	43,066	28.71	721	37.56
Asymmetric Cipher	independent	TEE_AsymmetricSignDigest	2,101,882	2,101.88	1,215,897	810.60	4,251,670	221,441.13
		TEE_AsymmetricVerifyDigest	5,205,865	5,205.87	2,693,087	1,795.39	2,859,461	148,930.28
Message Digest	independent	TEE_DigestUpdate	157,150	157.15	96,757	64.50	303	15.78
		TEE_DigestDoFinal	144,743	144.74	90,009	60.01	256	13.31

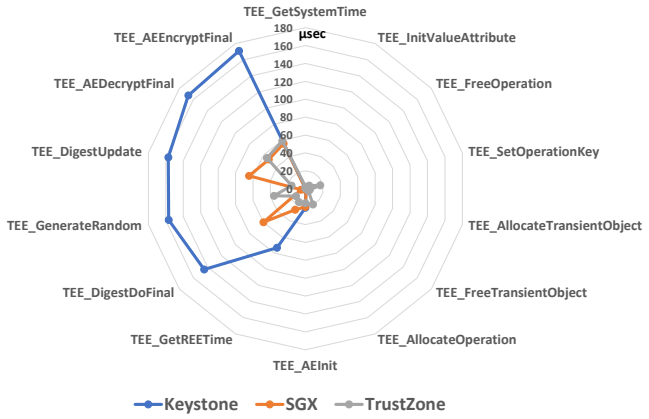

Fig. 4. Comparison of API Performance on Keystone, SGX, and TrustZone (µ seconds).

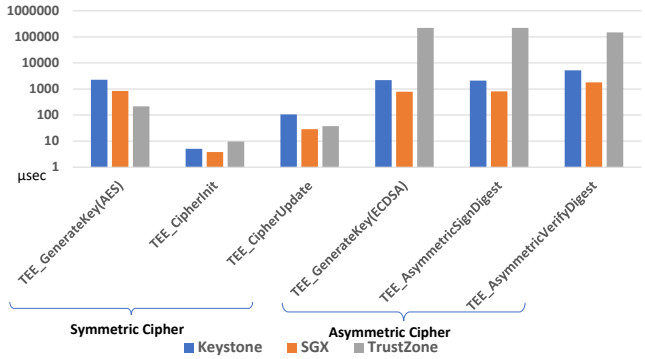

Fig. 5. Comparison of Symmetric and Asymmetric Cipher API Performance on Keystone, SGX, and TrustZone (µ seconds).

operations. They indicate that write operation does not use delayed writing but confirm the writing because the writing from TEE to the file system on host OS must be reliable. On the other hand, `TEE_CreatePersistentObject` is not stable on each TEE. This result is caused by extremely slow file creation. We guess it depends on the I/O scheduling on the host OS.

C. Performance of Edger

Another performance problem is the wrapper code generated by the EDL. The keystone has its original keyedge. The oeedger8r is experimentally ported to keystone by us before the keyedge becomes available.

Table V presents the API performances which use keyedge or oeedger8r on Keystone. We omit APIs for secure storage because the performances are more affected by I/O scheduling than by EDL codes. The results include the API's intrinsic time consumption and the overhead caused by EDL. Therefore, the difference column shows the performance difference between keyedge or oeedger8r. The results indicate that the keyedge is slower than oeedger8r.

We imagine that the principal difference came from that the wrapper codes, generated by the keyedge with **flatcc**, consume more time than those generated by the oeedger8r. Although flatcc can do very flexible serialization/deserialization safely, it is unfavorable for the simple data.

`TEE_GenerateKey` uses the same code of `TEE_GenerateRandom` because they require the same procedure to get a random number on the host OS. Therefore, the differences show almost the same results.

`TEE_GetREETime` shows that the difference is log time comparing with the total consumption time. It indicates the overhead of EDL is big for short time APIs. The developer should take care of these security overheads.

TABLE IV
CYCLE COUNT AND TIME FOR SECURE STORAGE APIS ON RISC-C KEYSTONE, ARM TRUSTZONE AND INTEL SGX

API	Stat	Keystone		SGX		TrustZone	
		cycle	µ-sec	cycle	µ-sec	cycle	µ-sec
TEE_CreatePersistentObject	ave	16,716,215	16,716.22	776,794	517.86	18,618,252	969,700.61
	min	1,452,444	1,452.44	660,800	440.53	2,058,475	107,212.24
	max	651,887,483	651,887.48	952,960	635.31	244,451,700	12,731,859.38
	stdev	77,025,419	77,025.42	34,829	23.22	37,674,290	1,962,202.61
TEE_OpenPersistentObject	ave	1,034,655	1,034.65	346,975	231.32	132,915	6,922.68
	min	1,025,455	1,025.46	301,212	200.81	131,095	6,827.86
	max	1,058,252	1,058.25	461,378	307.59	134,318	6,995.73
	stdev	4,297	4.30	44,368	29.58	732	38.11
TEE_WriteObjectData + TEE_CloseObject	ave	935,525	935.52	502,299	334.87	5,822,460	303,253.13
	min	484,921	484.92	399,840	266.56	1,036,901	54,005.26
	max	7,614,910	7,614.91	551,044	367.36	95,303,901	4,963,744.84
	stdev	1,445,705	1,445.71	23,248	15.50	15,187,445	791,012.78
TEE_ReadObjectData + TEE_CloseObject	ave	531,781	531.78	153,518	102.35	15,117	787.33
	min	528,143	528.14	149,796	99.86	14,520	756.25
	max	552,508	552.51	184,588	123.06	15,601	812.55
	stdev	2,728	2.73	4,007	2.67	245	12.75

TABLE V
API PERFORMANCE WITH DIFFERENT EDL ON KEYSTONE

API	keyedge µ-sec	oeedger8r µ-sec	Difference µ-sec
TEE_GenerateRandom	156.64	33.13	123.52
TEE_GetREETime	73.10	30.59	42.51
TEE_GenerateKey(AES)	2,262.15	2,135.78	126.37
TEE_GenerateKey(ECDSA)	2,199.13	2,068.64	130.48

VI. SECURITY ANALYSIS

Effect of EDL: The library implementation is based on a TEE's SDK which has EDL. The code for communication between TA and normal application is generated to prevent pointer and buffer misuse. It can prevent boomerang attack [20] and much safer than the normal GP implementation. However, the paper [26] showed that EDLs are not perfect. Our implementation allows to select the EDL on RISC-V (keyedge or oeedger8r), and the user can change the EDL when an EDL's problem is found.

Debugging: TA Debugging on our GP library depends on the based SDK, although trusted OS based implementation follows the GP debug specification [12]. Our GP library follows the error message issued from the based SDK, which may cause trouble. The unification is future work.

Effect of Optimization: The optimization options to build a library and trusted OS were default. If we used much higher optimization, the results would be different. Especially if we selected the optimization to use vector instructions, the results would be better. Availability is an important factor for security and the analysis of the effect of optimization is future work.

VII. CONCLUSION

The library-based implementation of GlobalPlatform TEE Internal API shows the typical design and implementation issues depend on the underlying TEE SDK. The separation of architecture-dependent and independent APIs makes clear the design choice. The evaluation exhibited that library implementation and trusted OS implementation showed a small performance difference in general. The important factors for performance are the implementation of cryptographic algorithms and I/O scheduling. The evaluation also shows the importance of EDL not only for security but also for performance.

ACKNOWLEDGMENTS

This paper is based on results obtained from a project, JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

[1] O. H. Alhazmi and Y. K. Malaiya, "Quantitative vulnerability assessment of systems software," in *Reliability and Maintainability Symposium (RAMS)*, 2005.
[2] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *computers & security*, vol. 26, no. 3, pp. 219–228, 2007.
[3] ARM. Mbed Crypto library. [Online]. Available: https://github.com/ARMmbed/mbed-crypto
[4] M. Bailleu, D. Dragoti, P. Bhatotia, and C. Fetzer, "TEE-Perf: A Profiler for Trusted Execution Environments."
[5] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, S. Devadas *et al.*, "MI6: Secure Enclaves in a Speculative Out-of-Order Processor," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.
[6] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.
[7] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *USENIX Security Symposium (USENIX Sec)*, 2016.
[8] ——, "Secure processors part I: background, taxonomy for secure enclaves and Intel SGX architecture," *Foundations and Trends in Electronic Design Automation*, vol. 11, no. 1-2, 2017.
[9] ——, "Secure processors part II: Intel SGX security analysis and MIT sanctum architecture," *Foundations and Trends in Electronic Design Automation*, vol. 11, no. 3, 2017.
[10] S. Dambra, L. Bilge, and D. Balzarotti, "SoK: Cyber Insurance–Technical Challenges and a System Security Roadmap," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.
[11] GlobalPlatform. [Online]. Available: https://globalplatform.org
[12] ——. GlobalPlatform API Archives. [Online]. Available: https://globalplatform.org/specs-library/

[13] Google. Asylo: an Open, Flexible Framework for Enclave Applications. [Online]. Available: https://asylo.dev/
[14] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, M. Grace, R. Padhye, C. Lemieux, K. Sen, L. Simon, H. Vijayakumar *et al.*, "PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation," in *USENIX Security Symposium (USENIX Sec)*, 2020.
[15] HexFive. MultiZone Secure IoT Stack. [Online]. Available: https://github.com/hex-five/multizone-secure-iot-stack/
[16] Intel. Intel Platform Developer Kit for Software Guard Extensions. [Online]. Available: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html
[17] D. Lee. Keystone SDK. [Online]. Available: https://github.com/keystone-enclave/keystone-sdk
[18] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments," in *European Conference on Computer Systems (EuroSys)*, 2020.
[19] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, "Keystone: A Framework for Architecting TEEs," *arXiv*, 2019.
[20] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments," in *Network and Distributed System Security Symposium (NDSS)*, 2017.
[21] Microsoft. Open Enclave SDK. [Online]. Available: https://openenclave.io/sdk/
[22] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, 2019.
[23] Raspberry Pi Foundation. Raspberry Pi 3 Model B+. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
[24] A. Rosenbaum, "Trusted Execution Environments," *Master thesis, Computer Science Department, Technion*, 2019.
[25] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2008.
[26] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes," in *Computer and Communications Security (CCS'19)*, 2019.
[27] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves."
[28] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V," in *Network and Distributed System Security Symposium (NDSS)*, 2019.
[29] wolfSSL. wolfSSL embedded SSL library. [Online]. Available: https://github.com/wolfSSL/wolfssl