Received August 14, 2021, accepted August 31, 2021, date of publication September 13, 2021, date of current version October 5, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3112202

TS-Perf: General Performance Measurement of Trusted Execution Environment and Rich Execution Environment on Intel SGX, Arm TrustZone, and RISC-V Keystone

KUNIYASU SUZAKI[1,2], (Member, IEEE), KENTA NAKAJIMA[2], TSUKASA OI[2], AND AKIRA TSUKAMOTO[1]

[1]National Institute of Advanced Industrial Science and Technology (AIST), Tokyo 135-0064, Japan
[2]Technology Research Association of Secure IoT Edge Application Based on RISC-V Open Architecture (TRASIO), Tokyo 101-0022, Japan

Corresponding author: Kuniyasu Suzaki (k.suzaki@aist.go.jp)

This work was supported by the New Energy and Industrial Technology Development Organization (NEDO) under Project JPNP16007.

ABSTRACT A trusted execution environment (TEE) is a new hardware security feature that is isolated from a normal OS (i.e., rich execution environment (REE)). The TEE enables us to run a critical process, but the behavior is invisible from the normal OS, which makes it difficult to debug and tune the performance. In addition, the hardware/software architectures of TEE are different on CPUs. For example, Intel SGX allows user-mode only, although Arm TrustZone and RISC-V Keystone run a trusted OS. In addition, each TEE has each SDK for programming. Each SDK offers own APIs and makes difficult to write a common program. These features make it difficult to compare the performance fairly between TEE and REE on different CPUs. To obtain precise performance and behavior in TEE, we propose TS-perf which is a compiler-based performance measurement method. TS-perf accesses the hardware timestamp counter in TEE as well as REE and keeps a precise log. The codes for measurement are inserted in a TEE binary by the compiler options (i.e., profile option, constructor, and destructor). Furthermore, we utilize the separate compilation technique, and the same benchmark binary is used for a fair comparison between TEE and REE. The architecture of TS-perf is general and implemented for three TEE architectures (Arm TrustZone, Intel SGX, and RISC-V Keystone). TS-perf measures the performance of GlobalPlatform's TEE internal APIs, matrix multiplication, memory access, and storage access. The comparisons show the difference in performance between TEE and REE and the unusual behavior of trusted applications (TAs).

INDEX TERMS Trusted execution environment (TEE), rich execution environment (REE), performance measurement, Arm TrustZone, Intel SGX, RISC-V Keystone.

I. INTRODUCTION

Since recent OSs support many hardware/software functions, they have become very large and complex and cannot escape vulnerabilities [1], [2]. To avoid these vulnerabilities, current CPUs offer an isolated execution environment for critical processing (i.e., TA: trusted application). The hardware isolation is called trusted execution environment (TEE) and is independent of a normal OS, namely, rich execution environment (REE). The TEE is equipped on popular CPUs (e.g., Arm TrustZone [3], [4], Intel SGX [5]–[8], AMD SEV [9]) as well as experimental CPUs (e.g., RISC-V based TEE; Sanctum [10], MI6 [11], MultiZone [12], TIMBER-V [13], Keystone [14], [15], Hector-V [16], CURE [17], and uTango [18]).

Most TEE hardware architectures are implemented by changing the `state` from REE to TEE on a core, which means that the same core is used in REE and TEE. It seems to show the same performance in REE and TEE, but each feature/limitation of each TEE architecture affects performance. For example, Intel SGX offers 96MB encrypted memory and user-mode (ring 3) only for a TA, whereas Arm TrustZone and RISC-V Keystone offer supervisor-mode to run a trusted OS on normal memory. (For example, Arm TrustZone is used by

The associate editor coordinating the review of this manuscript and approving it for publication was Xiali Hei.

many smartphones with a vendor specific trusted OS; KNOX for Samsung [19]–[21], RTOSck for Huawei [22], QSEE for Qualcomm [23], [24], etc. RISC-V Keystone also offers some choices: eyrie [14], [15], seL4 [25], etc.)

In addition, each TEE has a software development kit (SDK) for programming a TA. Unfortunately, the abstraction depends on SDK, and APIs for TA are not standardized (namely, there is no POSIX-like standard). These features make it difficult to write a portable program for TEE.

On the other hand, availability is an important factor for security, but the current TEE does not support adequate performance measurement tools because TEE hides the behavior from the normal OS. Therefore, TA debugging and performance tuning are not easy. To solve this issue, some approaches have been proposed [26]–[34], but they include high overhead or dependency of special hardware. In addition, the previous approaches are not based on the same reliable time counter between TEE and REE, and the performance comparison is not accurate.

To solve these problems, we have implemented the portable GlobalPlatform TEE internal API library for different TEE architectures, but the performance details have not been measured because we lacked a suitable measurement method. In addition, we cannot confirm that the performance in TEE is the same as that in REE. This has given us motivations to establish a fair performance comparison that runs the same binary in TEE and REE and measures the performance with the same time resource in both environments.

This paper proposes the precise measurement method `TS-perf` which utilizes a CPU hardware timestamp counter in TEE and REE, and GCC [35] compiler options to insert the codes for measurement. TS-perf is combined with the separate compilation technique and makes it possible to run the same binary in TEE and REE to enable fair performance comparison. The performance measurement results show some unusual behaviors (e.g., strange core change and uncertain CPU load), and we confirm that they come from the difference in TEE implementations or a bug.

Contributions and Challenges:

1) **General Performance measurement based on the hardware timestamp in TEE and REE:** We design `TS-perf` which is a compiler-based performance measurement method that can be applied in many TEE implementations. `TS-perf` obtains the hardware timestamp in TEE as well as in REE. The codes for measurement are inserted by the GCC compiler options (i.e., `profile option`, `constructor`, and `destructor`). The result is reported to the REE after the full logging process, which does not cause runtime overhead. `TS-perf` is implemented for three TEE architectures (i.e., Arm TrustZone, Intel SGXv2, and RISC-V Keystone).
2) **Comparison of the same benchmark in TEE and REE:** In many cases, different compiler options are used for TEE and REE applications and result in different binaries. For a fair comparison, we utilize the `separate compilation` technique, and the same object is linked to the benchmark binaries in TEE and REE. The performance is measured by the same time resource between TEE and REE using `TS-perf`, which enables fair performance comparison on a state-changing style TEE. The results can show the precise differences between TEE and REE on different CPUs.
3) **Implementation benefits:** The techniques used by `TS-perf` (i.e., hardware timestamp, compiler options) and the separate compilation are general techniques and can be applied to many TEE implementations. The ability is shown by the implementation for Arm Cortex-A (TrustZone), Intel x86-x64 (SGX), and RISC-V U540 (Keystone).
4) **TEE behavior is compared with the view of the normal OS:** The behavior of TA is monitored from the view of the normal OS and compared with the results of `TS-perf`. We confirm some unusual behaviors (i.e., core change and uncertain CPU load) and find an interrupt-handler bug between REE and TEE.

The remainder of this paper is organized as follows. Section II describes the background knowledge. Section III depicts the design of `TS-perf` and the separate compilation. Section IV describes the implementation of `TS-perf`, and section V shows the measured performance and behavior. Section VI describes some related topics, and section VII concludes this paper.

II. BACKGROUND

This section describes the background of the TEE architecture, time counter, common TEE programming API, and related works for performance and behavior measurement in TEE.

A. TEE ARCHITECTURE

The three TEE architectures used in this paper are described. These TEEs are implemented by changing the `state` of the CPU cores.

1) ARM TrustZone

Since smartphones, game machines, and set-top boxes use Arm Cortex-A TrustZone [3], [4][1] for critical processing, it is the most popular TEE. TrustZone offers 2 world view model, i.e., the secure world (i.e., TEE) and the normal world (i.e., REE). The world (namely, the `state` of REE and TEE) is changed by the `SMC` (secure monitor call) instruction. Each world has user- and supervisor-mode and runs applications on a kernel (i.e., trusted OS or normal OS). Many trusted OSes on TrustZone(e.g., QSEE [23], [24], OP-TEE [36]) offer the APIs defined by GlobalPlatform [37], [38] for TA programming. Some APIs require the help of a normal OS (e.g., secure storage).

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

The memory allocation for TEE is flexible, but most implementations allocate the limited memory at the boot time to keep a small trusted computing base (TCB). When an interrupt is issued, a trusted OS can handle it if the interrupt is caused by the secure world's peripheral. However, many interrupts are passed to a normal OS with a context switch because they are issued for the normal world.

2) INTEL SGX

Intel Software Guard Extensions (SGX) [5]–[8] offers a single address model of TEE, named `enclave`. SGX allows a TA as a part of a normal application on a host OS, and the TA runs on the user-mode (ring 3) in an enclave which is the `state` of TEE. An enclave is created dynamically, and the TA is loaded on it. The TA is implemented as a shared library offered by Intel SDK. When an interrupt is issued, the processing is passed to the normal OS in REE. The total memory for enclaves is defined by UEFI and reserved at power-on. The maximum size is fixed (128MB is reserved on SGX version 1, and 256MB is reserved on version 2). The memory region is encrypted with the key generated at power-on.

Intel SDK includes the enclave definition language (EDL) named `edger8r`, which offers glue codes for secure communications between the normal application and the TA (i.e., `OCALL` from the TA to the application and `ECALL` from the application to the TA). The glue codes check the region of the pointer and the size of the buffer. They wrap the edge/boundary and serialize/deserialize the arguments/results. The glue codes are crafted with formal verification to reduce attack surfaces. An exhaustive analysis of the EDL is described in [39].

3) RISC-V KEYSTONE

RISC-V is an open instruction set architecture (ISA) and has some TEE implementations [10]–[18]). This paper utilizes Keystone because it is open-source and runs on a real CPU (SiFive's Unleashed board). Keystone, a mixed architecture of Arm TrustZone and Intel SGX, can create a TEE dynamically as SGX, although TrustZone offers only one TEE at boot time. The TEE is named `enclave` as SGX but 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 dynamically from REE (i.e., Linux) using the physical memory protection (PMP) mechanism. The dispatched memory is sanitized and used for critical processing. The PMP also manages the `state` of cores and changes from REE to TEE. In the same manner as Intel SGX, Keystone offers the EDL named `keyedger`, and the glue codes protect the OCALL.[2]

B. TWO TYPES OF TIME COUNTERS

Current computers keep two types of time. One is the hardware timestamp, and the other is the system clock which indicates the global clock (i.e., calendar clock).

[2]Current Keystone has not implemented ECALL yet.

TABLE 1. Hardware timestamp counter, reading instruction, and frequency for each architecture.

	Counter	Instruction	Frequency (MHz)
Intel x86-64 (Pentium J5005)	TSC	rdtsc	1,500
Arm Cortex-A (Cortex-A53)	CNTVCT_EL0	mrs	19.2
RISC-V RV64 (SiFive U540)	HPM	rdcycle	1,000

1) TIMESTAMP CYCLE

The hardware timestamp counts the number of the internal processor clock cycle. The timestamp counter is a special hardware resource in a CPU, and access is allowed by special instructions. For example, the Arm CPU's counter-timer physical count register `CNTPCT_EL0` is allowed for the supervisor only, but the counter-timer virtual count register `CNTVCT_EL0` is allowed for the user. In addition, some CPUs can change their CPU speed frequency to reduce power consumption, but the internal processor clock cycle is not affected, and the timestamp counter remains monotonic.

Table 1 shows the timestamp counters of the target architectures of this paper. They are Intel x86-64's timestamp counter `(TSC)` with `rdtsc` instruction, Arm Cortex-A's counter-timer virtual count register `(CNTVCT_EL0)` with `mrs` instruction, and RISC-V's hardware performance monitor `(HPM)` with `rdcycle` instruction. The table also includes the frequency of the hardware timestamp of the CPU used in this paper. Arm Cortex-A53 has a slower hardware timestamp than the CPU speed frequency (1,400 MHz). Thus, the resolution is not high but is sufficient to measure a function-level amount of code, as shown in Section V.

2) SYSTEM CLOCK

The system clock indicates the global clock, which is used for verifying certificates and logging. In general, the source of the system clock is based on an external peripheral named real-time clock (RTC) at boot time. The RTC has a battery power supply and keeps working even if the CPU power is down.

The system clock is obtained by a system call `gettimeofday()` on Linux. If `gettimeofday()` is implemented purely, it accesses the RTC. However, much time is needed to access the external peripheral. To reduce the access overhead, the initial value of the system clock is obtained from the RTC, and the system clock is maintained by other clock resources (e.g., timestamp counter). In addition, most implementations of `gettimeofday()` use a virtual dynamic shared object (VDSO) and omit the context switch of a system call.

C. COMMON TEE PROGRAMMING API

In general, each TEE has an SDK for its TA programming. For example, Intel offers SGX-SDK [5], and RISC-V Keystone also offers Keystone SDK [40]. To solve this problem,

TABLE 2. Comparison of related works.

	Target	Time Source
Gjerdrum et al. [26]	SGX	timestamp outside TEE
SGX-Perf [27]	SGX	timestamp outside TEE
TEE-Perf [28]	General (Implemented on SGX)	software counter inside TEE
TEE-Mon [29]	SGX	timestamp inside TEE
TS-Perf	SGX, TrustZone, Keystone	timestamp inside TEE

we implemented the portable library of GlobalPlatform's TEE Internal API [41] because the specification is opened and widely used on smartphones.

On Arm TrustZone, the trusted OS `OP-TEE` offers GlobalPlatform's TEE Internal API already, and we implemented the portable library for Intel SGX and RISC-V Keystone. Intel SGX has no supervisor mode, and RISC-V Keystone offers `eyrie` runtime at the supervisor level, which provides limited API for a TA. GlobalPlatform's TEE Internal API includes approximately 300 APIs with many arguments that include many cipher suites. We selected notable APIs for common applications.

We divide the GlobalPlatform APIs into 2 categories: CPU-independent and CPU-dependent. CPU-independent APIs are cryptographic functions and can be implemented easily as a portable library, whereas CPU-dependent APIs are functions related to secure storage, timer, and random number. These APIs must be customized for each architecture but are also implemented as a library.

D. RELATED WORKS

The importance of TEE performance measurement has been recognized, and some methods have been proposed.

Table 2 summarizes the related works. Gjerdrum *et al.* [26], SGX-Perf [27], and TEEMon [29] are Intel SGX specific solutions and cannot be applied to other architectures. Gjerdrum's and SGX-Perf are early approaches and use the timestamp outside TEE. Their main target is the performance of SGX primitive functions (e.g., context switches) and does not provide a method to measure the functions of a TA. TEEMon is a real-time performance monitoring framework that uses the timestamp inside TEE, but the overhead is reported to be high from 5% to 17%.

TEE-Perf [28] is a general perf implementation but assumes a recorder process that occupies a core along with the perf process. The recorder process has a software counter on a shared memory, which is incremented by an infinite loop. The perf process obtains the number of the software counter via shared memory. TEE-Perf can be applied to REE, but the implementation is redundant and inaccurate (the overhead is reported to be up to 5.7× higher than that for Linux perf).

On Arm TrustZone, some trusted OSs offer measurement functions because they are based on the GlobalPlatform APIs which include time measurement (i.e., `TEE_GetREETime()` and `TEE_GetSystemTime()`). Furthermore, OP-TEE offers gprof [30], [31] to measure the performance of each function. The paper [32] customizes the OP-TEE to measure the performance of the stress test benchmark; STRESS-NG [42]. They are useful but are not generic to enable comparison to other architectures.

Some CPU vendors offer hardware assistance to obtain the performance information. Intel Processor Trace (PT) [33] and Arm CoreSight [34] are well-known mechanisms and integrated current perf tools, but they are disabled by default on SGX and TrustZone.

`TS-perf` measures performance using a general hardware timestamp counter inside TEE and REE, and the results are precise. We offer the same APIs on different TEE architectures and make it possible to compare the performance between different architectures.

III. DESIGN

This section describes two design issues for `TS-perf` and the method of comparing REE and TEE.

A. DESIGN OF TS-PERF

`TS-perf` is a compiler-based performance measurement method that obtains the time from the hardware timestamp counter in TEE and REE. The time data are saved in memory during performance measurement because the writing of data involves high overhead. `TS-perf` writes the time data in memory to a file after the measurement.

To implement these functions, `TS-perf` has three challenges. The first concerns the access to the hardware timestamp counter in TEE and measuring the time before and after a function. The second is the memory allocation for logging in TEE. The third is the writing of data from TEE to a file in REE. `TS-perf` solves these challenges using the features of the `GNU Compiler Collection (GCC)` [35] (i.e., `profile option`, `constructor`, and `destructor`).

The means of accessing the hardware timestamp counter in TEE is dependent on each architecture. On Intel SGX, `TS-perf` is implemented for SGX version 2 (SGXv2) with `rdtsc` instruction because SGX version 1 (SGXv1) does not allow access to TSC in user-mode. SGX2 is offered for limited CPUs only, but we choose an available machine. Accessing the timestamp counter in SGX is not our contribution, but we utilize it for fair performance comparison. Arm TrustZone and RISC-V Keystone allow access to their TSC in user-mode (i.e., `CNTVCT_EL0`. and `HPM` with `rdtsc` and `rdcycle` instruction, respectively). The code to obtain the timestamp must be inserted on the top and bottom of a target function. Fortunately, the `profile option` of GCC offers this feature, and `TS-perf` utilizes it.

The second challenge concerns how to keep the time on memory in TEE. The memory region must be assigned before measurement because it reduces the runtime overhead. Fortunately, the `constructor` of GCC offers the mechanism to insert code before the main program. `TS-perf` utilizes the `constructor` to reserve memory for logging.

The third challenge is reporting log data to REE after measurement because access to REE includes heavy overhead. The `destructor` of GCC enables insertion of a code after the main function, and `TS-perf` utilizes it. The way to save data from TEE to a file in REE depends on the TEE implementation. `TS-perf` follows the manner suitable for each architecture.

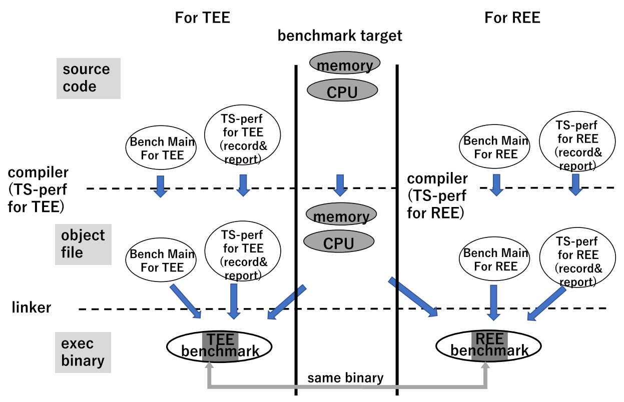


FIGURE 1. Separate compilation for TEEE and REE benchmark.

B. DESIGN OF A METHOD FOR COMPARING REE AND TEE

`TS-perf` enables precise performance measurement based on the same timestamp in TEE and REE, but it is insufficient because the same binary cannot be run in TEE and REE in general. The reason is that the system call is different, and a TA must link special libraries for TEE execution. In addition, the compiler options are different in many cases. If link time optimization (LTO) is enabled, the functions may be optimized differently. To solve this problem, we utilize the `separate compilation` technique.

Figure 1 shows the overview. The functions for measuring are saved in an original source file and compiled to an object file (in Figure 1, `memory` and `CPU` are the measuring targets). The benchmark main and TS-perf codes are prepared for TEE and REE because they depend on these environments. The benchmark main code includes the function to call the measuring target, and the TS-perf code uses GCC profiling. `TS-perf` offers measuring codes for TEE and REE. Hence, TS-perf does not use the GCC original `prof` because of the objective of fair comparison.

The measuring target object file is linked to the benchmark main and TS-perf object and creates a benchmark binary. The same measuring target binaries are thus used in REE and TEE.

Unfortunately, this technique is not applied to the function that uses TEE architecture-dependent APIs (e.g., secure storage). In that case, the source code is written as similar as possible and compiled with the same options.

IV. IMPLEMENTATION

`TS-perf` is implemented for Arm TrustZone, Intel SGX, and RISC-V Keystone. The implementation in `TS-perf` consists of three steps on `GCC` [35]. `TS-perf` is also implemented for REE, but this section concentrates on TEE because the implementation is almost the same and simple.

Step 1 (Compiler Phase): The profile option of `GCC` requires the compile flag to build an object for measuring functions. The flag is `-finstrument-function`, which inserts a code at the top and the bottom of every function. These inserted codes access the timestamp counter from TEE on each architecture.

`TS-perf` also requires the codes for preparing the log buffer and reporting the log to REE (i.e., normal OS). These codes must be executed before or after the main part of TA execution. The `GCC` compiler offers `__attribute__((constructor))` and `__attribute__((destructor))` to insert code at the start and end of the main function. However, the linker script for the TA is not compatible on each architecture, and it is not easy to insert arbitrary codes.

Fortunately, each TA has its own entry point which enables the insertion of arbitrary codes before and after the TA. The entry points are `eapp_entry`, `ecall_ta_main` and `TA_InvokeCommandEntryPoint` on RISC-V Keystone, Intel SGX, and Arm TrustZone, respectively. The code for obtaining a 64KB log buffer is inserted before the entry point. Its size is fixed for simplicity, and it is large enough to profile on each function. The code to report the log data to REE is inserted after the entry point. The implementation depends on each architecture described as follows.

Step 2 (Recorder Phase): During the TA execution, the address of the function and the timestamp counter value of each enter/exit are logged into the buffer. After the TA is finished, the buffer data must be written to the log file in REE. (Note: `TS-perf` runs a log-saving code in REE and writes the log file using the system call of Linux.)

In `TS-perf` in TEE, the `__profiler_map_info` and `__profiler_unmap_info` functions are registered in the entry point of each TEE and play an important role in logging. The `__profiler_unmap_info` function is inserted before the main function of TA by `__attribute__((constructor))` and prepares the log buffer (64KB). The `__profiler_unmap_info` function is inserted after the main function of TA by `__attribute__((constructor))` and saves the log to a file in REE (i.e., Linux).

The implementation of the `__profiler_unmap_info` function is different on each TEE because it needs to collaborate with Linux. In Keystone and SGX, OCALL functions are used to write the log buffer to the file in REE. On the other hand, in TrustZone (namely, OP-TEE), the log data are moved into the shared buffer. After the end of the TA, it is written to the file in REE.

In RISC-V Keystone, the entry point is the `eapp_entry` function which should be added with the `EAPP_ENTRY` keyword with The `__profiler_unmap_info`. The `__profiler_unmap_info` uses OCALL functions such as open, write, and close for the log file on Linux (in Code 1).

```
#define EAPP_ENTRY \\
   _attribute_((_section_(".text._start")))
void EAPP_ENTRY eapp_entry()
  {main();
    __profiler_unmap_info();
    EAPP_RETURN(0);}

  // in __profiler_unmap_info()
  int fd = ocall_open_file(
    LOG_FILE, O_RDWR | O_CREAT,
                   (mode_t)0600);
    ocall_write_file(fd, ptr, sz);
    ocall_close_file(fd);
```

Code 1. Saving log in REE by OCALL on RISC-V Keystone.

```
  void ecall_ta_main(void) {
    int retval;
    main();
    __profiler_unmap_info();
    return;
  }

  // in __profiler_unmap_info()
  sgx_status_t val;
  int fd;
  val =
    ocall_open_file(&fd, LOG_FILE,
      O_RDWR | O_CREAT, (mode_t)0600);
  ocall_write_file(&val, fd, ptr, sz);
  ocall_close_file(&val, fd);
```

Code 2. Saving log in REE by OCALL on RISC-V Keystone.

In Intel SGX, the `__profiler_unmap_info` is also registered at the SGX entry point `ecall_ta_main`. The OCALL functions are used in the same way as Keystone, but the arguments are not the same (in Code 2).

In Arm TrustZone, a secure OS `OP-TEE` manages a TA. OP-TEE has no OCALL as Keystone and SGX, and we have to write a program to share log data. We prepare the buffer both in REE and in TEE explicitly to share the log data. In REE, the shared buffer is allocated with the output flag. The buffer is conveyed to the TA by `TEEC_InvokeCommand`, which starts the TA in TEE (in Code 3).

```
// in REE
#define BUF_SIZE 65536
memset(&op, 0, sizeof(op));
op.paramTypes =
 TEEC_PARAM_TYPES(
     TEEC_MEMREF_TEMP_OUTPUT,
   TEEC_NONE, TEEC_NONE, TEEC_NONE);
static char buf[BUF_SIZE];
op.params[0].tmpref.buffer = (void*)buf;
op.params[0].tmpref.size = BUF_SIZE;

// Dive into TEE when the TA is executed
res = TEEC_InvokeCommand(
    &sess, TA_REF_RUN_ALL,
    &op, &err_origin);
```

Code 3. Getting Buffer between TEE and REE on OP-TEE/Arm TrustZone.

Inside the TA, the `__profiler_unmap_info` function is also registered by `TA_InvokeCommandEntryPoint`.

```
// In TEE
  TEE_Result TA_InvokeCommandEntryPoint(
    void *sess_ctx,
    uint32_t cmd_id,
    uint32_t param_types,
    TEE_Param params[4])
  { main();
    __profiler_unmap_info(
      params[0].memref.buffer,
      &params[0].memref.size);
      return TEE_SUCCESS;}

  // This is the profiler logged data.
  extern struct __profiler_header
                    * __profiler_head;
  void __profiler_unmap_info(char *buf,
                       size_t *size)
  { memmove(buf, __profiler_head, sz);}
```

Code 4. Transferring log from TEE to REE on OP-TEE/TrustZone.

```
  // In REE. Save the data as a file.
  int fd = open(LOG_FILE, O_RDWR |
              O_CREAT, (mode_t)0600);
  write(fd, op.params[0].tmpref.buffer,
              BUF_SIZE);
  close(fd);
```

Code 5. Log Saving in REE on Linux/Arm.

In TEE, after the main program, the `__profiler_unmap_info` function is called (in Code 4). In the `__profiler_unmap_info` function, the log data are simply moved to the shared buffer.

When the TA terminates, the `TEEC_InvokeCommand` function returns in REE. The buffer is packed with the log data for each function. The log data are saved with POSIX-compliant functions such as open, write, and close in REE (in Code 5).

Step 3 (Analyzer Phase): The performance result is saved as a binary file. The analysis tool parses the data, organizes into a readable format, and compares the figure between the different architectures.

TABLE 3. Target machines (The intel pentium CPU does not include hyperthreading.).

	CPU	Core	Mem (GB)	REE	TEE
Raspberry Pi3 B+ [43]	Cortex -A53	4	1	Linux 4.14.56	TrustZone (OP-TEE 3.8.0)
Intel NUC 7BJYH [44]	Pentium J5005	4	8	Linux 5.3.0	SGXv2 (SDK v2.8)
SiFive Unleashed [45]	U540	4	8	Linux 4.15.0	Keystone v0.3 (Eyrie runtime)

V. EVALUATION

`TS-perf` measured some types of benchmarking in TEE and REE on three different architectures listed in Table 3. The CPU speed frequency is fixed at 1,000 MHz by `cpufreq-set` to prevent an automatic change on Intel x86-64. However, we preformed evaluations on 1,400 MHz Arm because the Raspberry Pi3 B+ offers 600

TABLE 4. Timestamp Counts and Time (μ-seconds) for `TEE_GetREETime()` and `TEE_GetSystemTime()`.

	Arm TrustZone	Intel SGX	RISC-V Keystone
Timestamp	19.2MHz	1,500MHz	1,000MHz
CPU Speed	1,400MHz	1,000MHz	1,000MHz
`TEE_GetREETime()`			
Ave.	188 (9.79)	27,128 (18.09)	42,153 (42.15)
Max	753 (39.22)	195,660 (130.44)	79,970 (79.97)
Min	175 (9.11)	26,326 (17.55)	41,312 (41.31)
Stdev	52 (2.71)	9,832 (6.55)	2,778 (2.78)
`TEE_GetSystemTime()`			
Ave.	4 (0.21)	552 (0.37)	486 (0.48)
Max	17 (0.89)	2,826 (1.88)	937 (0.94)
Min	3 (0.16)	538 (0.36)	382 (0.38)
Stdev	2.71 (0.14)	132 (0.09)	78 (0.08)

or 1,400 MHz frequency only. Each benchmark is measured 200 times, and the average is shown.

The evaluations aim to show (1) the accuracy and precision of performance measurement, (2) the difference in TEE implementation on different CPUs, and the difference between TEE and REE on the same CPU, and (3) the unusual behavior in TEE.

A. OVERHEAD FOR OBTAINING TIME

To show the accuracy of `TS-perf`, we measured the time functions of GlobalPlatform TEE Internal APIs: `TEE_GetREETime()` and `TEE_GetSystemTime()`. `TEE_GetREETime()` obtains the system clock from REE, and `TEE_GetSystemTime()` obtains the hardware timestamp in the user-mode of TEE, which is the same as that in `TS-perf`.

Table 4 shows the results. `TEE_GetREETime()` causes OCALL, and the average time is more than 15 μ-seconds on each architecture. On the other hand, the average time of `TEE_GetREETime()` is less than 0.5 μ-seconds on each architecture. Hence, the average time of `TEE_GetSystemTime()` is 30 times faster than that of `TEE_GetREETime()`. The maximum time and standard deviation in TEE_GetREETime() on the Arm TrustZone and Intel SGX were higher than those on the RISC-V Keystone. We speculate that the differences are caused by the complex hardware on Arm and Intel (e.g., cache hierarchy, branch prediction). The relative maximum time and standard deviation on `TEE_GetSystemTime()` are less than those on TEE_GetREETime(), but the absolute values of `TEE_GetSystemTime()` are shorter than those on TEE_GetREETime().

The time-related functions were measured by `TS-perf`, which uses the hardware timestamp as `TEE_GetSystemTime()`. The standard deviations were low; therefore, `TS-perf` is stable and accurate.

B. TEE AND REE BENCHMARKS

We use three original benchmarks because existing benchmarks are not suitable for TEE. They assume input/output or system calls that are not supported in TEE. In addition, we want to show a fair performance comparison between TEE and REE. The three benchmarks are CPU, memory, and storage intensive.

1) FEATURES OF BENCHMARKS

CPU Intensive: CPU-intensive benchmarks measure 25,000,000 multiplications of integers or double float numbers. They are simple arithmetic benchmarks and are assumed to have no difference in TEE and REE. The benchmarks utilize the separate compilation technique for a fair comparison. (Note: The iteration number is determined by the average elapsed time on all architectures.)

Memory Intensive: Memory-intensive benchmarks measure 1MB memory read/write access sequentially or randomly. The benchmarks may cause performance differences when the memory is encrypted. However, cache and branch prediction may hide the performance difference. The benchmarks utilize the separate compilation technique for a fair comparison. (Note: the memory size is decided to compare all architectures.)

Storage Intensive: Storage-intensive benchmarks measure the 1MB file read or write sequential access only because the current implementations of GlobalPlatform APIs for storage (i.e., `TEE_WriteObjectData()` and `TEE_ReadObjectData()`) do not allow random access. Read or write access occurs for each 32KB unit due to the TEE buffer size. Storage depends on the different API implementations in TEE and REE. Therefore, the separate compilation technique cannot be used.

2) RESULTS OF BENCHMARKS

Table 5 summarizes the results for TEE and REE, and Figure 2 visualizes the results for the CPU- and memory-intensive benchmarks.

CPU Intensive: The results show almost the same performance for the multiplication of integers and double float numbers in TEE and REE. These results are quite natural because TEE and REE run on the same core architecture. However, each architecture has a slight difference. Arm TrustZone shows that TEE is approximately 3% slower; this impact is the highest among the three CPUs. The maximum and minimum times did not have large differences, but the differences were stable. We hypothesize that there are the architectural differences, and analysis is left to future work.

Memory Intensive: The results show almost the same performance for sequential and random memory access in TEE and REE of Intel SGX and RISC-V Keystone. Arm TrustZone shows that TEE is slower, especially with 11% overhead on random access. Arm has a large impact on random memory access in TEE, and programmers should exercise caution.

We expected that SGX's memory encryption mechanism would cause performance degradation, but the results do not show this feature. We analyzed further performance on Intel, and we changed the memory size from 1MB to 32MB. Figure 3 shows the results. The sequential access

TABLE 5. Performance comparison between TEE and REE on Arm Cortex-A, Intel X86-64, and RISC-V U540.

	CPU(Integer)		CPU(double Float)		Mem(Sequential)		Mem(Random)		Storage(Read)		Storage(Write)	
	Arm Cortex A53 19.2MHz (CPU Speed 1,400MHz)											
	REE	TEE	REE	TEE	REE	TEE	REE	TEE	REE	TEE	REE	TEE
Ave.[cycles]	6,272,535	6,459,298	8,716,457	9,016,912	486,179	506,637	5,143,468	5,707,502	47,263	4,405,263	2,681,626	471,026,830
Ave.[μ-sec]	326.69	336.42	453.98	469.63	25.32	26.39	267.89	297.27	2.46	229.44	139.67	24,532.65
Ave.rate	1.030		1.034		1.042		1.110		93.283		288.733	
Stdev.[cycles]	3,215	4,591	5,060	5,880	987	896	12,304	12,799	1,362	5,714	4,274,413	215,509,389
max[cycles]	6,280,070	6,468,034	8,724,285	9,030,406	490,877	510,343	5,185,822	5,785,363	54,847	4,429,964	40,935,853	1,532,045,904
min[cycles]	6,268,932	6,451,412	8,705,560	9,008,798	484,439	504,542	5,116,873	5,680,809	44,228	4,392,517	108,843	287,636,682
	Intel Pentium J5005 1,500MHz (CPU Speed 1,000MHz)											
	REE	TEE	REE	TEE	REE	TEE	REE	TEE	REE	TEE	REE	TEE
Ave.[cycles]	527,743,603	529,163,580	979,360,413	982,030,247	33,360,804	33,452,354	79,849,326	80,238,753	1,012,201	23,595,244	11,023,572	13,802,220
Ave.[μ-sec]	351.83	352.78	652.91	654.69	22.24	22.30	53.23	53.49	0.67	15.73	7.35	9.20
Ave.rate	1.003		1.003		1.003		1.005		23.329		1.291	
Stdev.[cycles]	173,977	210,251	405,469	391,820	66,000	64,700	76,419	91,668	30,352	67,197	2,632,873	2,790,326
max[cycles]	528,291,872	529,963,012	980,433,320	983,339,596	33,875,652	33,838,626	80,153,750	80,587,376	1,225,080	23,973,296	25,835,106	28,438,830
min[cycles]	527,456,854	528,823,954	978,682,392	981,199,324	33,325,768	33,406,530	79,783,736	80,154,918	996,582	23,509,882	3,438,128	6,200,978
	RISC-V U540 1,000MHz (CPU Speed 1,000MHz)											
	REE	TEE	REE	TEE	REE	TEE	REE	TEE	REE	TEE	REE	TEE
Ave.[cycles]	625,684,260	625,165,903	1,150,806,936	1,175,295,298	52,206,674	52,221,679	146,735,124	146,651,856	4,049,370	1,377,719,027	962,966,946	1,252,208,504
Ave.[μ-sec]	625.68	625.17	1,150.81	1,175.30	52.21	52.22	146.74	146.65	4.05	1,377.72	962.97	1,252.21
Ave.rate	0.999		1.021		1.000		0.999		345.147		2.258	
Stdev.[cycles]	753,165	2,617	192,296	60,471	68,149	219,738	58,121	387,724	559,357	435,818	517,036,960	1,426,607,344
max[cycles]	636,108,921	625,200,640	1,153,495,142	1,175,559,332	53,065,196	53,878,933	147,235,233	148,038,808	6,172,090	1,378,781,668	2,852,414,990	20,301,373,780
min[cycles]	625,427,661	625,165,361	1,150,756,221	1,175,275,481	52,188,329	52,150,379	146,669,853	146,078,985	3,691,263	1,377,035,646	9,594,759	253,049,942

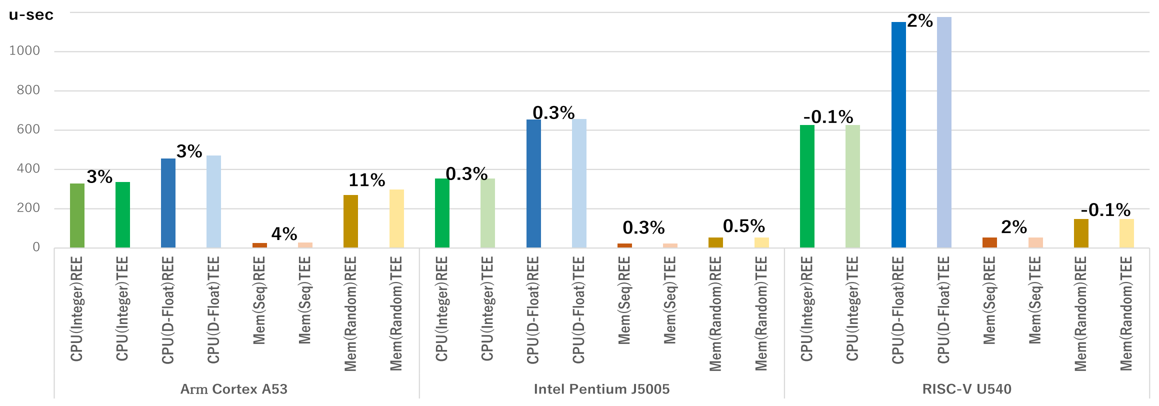


FIGURE 2. Performance comparison between TEE and REE on Arm Cortex-A, Intel X86-64, and RISC-V U540.

performance is almost proportional to the memory size, but the random accesses are slower for TEE than for REE. Table 6 shows the detailed results. The performance degradation is not clear until 4MB. We expect the same performances on small memory to be caused by the CPU cache because Pentium j5005 has a 4MB L2 cache. We also expect the degradation in random access in TEE to be caused by memory encryption because the effects of the cache are the same in REE and TEE. The overhead for memory encryption is exposed upon large memory random access.

Storage Intensive: The results show the difference between TEE and REE. As expected, the results were unstable because TEE requires OCALL to save the encrypted data in REE (i.e., Linux). On TrustZone, both read and write performance in TEE showed a large difference, perhaps because implementation is complex for OP-TEE in terms of file access.

TABLE 6. Memory access performance on SGX (cycle).

Size	REE Sequential	TEE Sequential	REE Random	TEE Random
1MB	33,336,127	33,467,479	79,909,846	80,257,343
2MB	66,810,658	67,048,488	160,035,078	160,736,098
4MB	134,636,496	136,106,088	584,069,737	863,116,053
8MB	267,627,753	270,311,064	1,435,650,340	4,430,440,190
16MB	536,002,394	541,325,427	3,079,191,221	12,800,353,655
32MB	1,071,971,620	1,081,301,082	6,725,818,165	28,174,547,437

SGX and Keystone cause OCALLs, which include glue code created by the EDL. We expected that the EDL affects the performance. However, the stability is not good, and the results cannot clearly show the effect of the EDL.

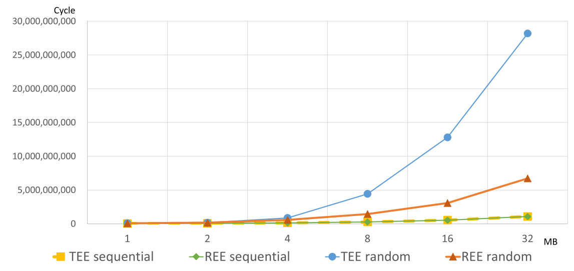


FIGURE 3. Memory Access Performance on SGX.

TABLE 7. The view of the TA from REE.

	Arm TrustZone	Intel SGX	RISC-V Keystone
Load Type	System	User	System
Load Drop from 100%	—	—	✓
Core Change	✓	✓	✓
Different Core	✓	—	—

C. COMPARING FROM THE VIEW OF REE

The behaviors of TEE benchmarks were monitored by `htop` on Linux (REE), which shows the load on each core. The results showed two unusual behaviors: (1) the TEE running core was changed, and (2) the core load did not remain at 100% even if a heavy benchmark was run. Table 7 summarizes the results.

1) CORE CHANGE

Because the core maintained a 100% CPU load, `htop` informed which core was used for the TEE benchmark. The CPU load category shown by `htop` was different in each TEE architecture: `system load` for TrustZone and Keystone and `user load` for SGX, which indicated the view of the TA from REE. In addition, `htop` showed that the 100% load core changed sometimes. This behavior was unusual.

We confirmed that the TA's core was changed when the normal application in REE was changed by the `taskset` command, which can designate the running application core. These results indicate that the Linux scheduler changes the process's core even if the process uses TEE. Therefore, the current Linux scheduler does not recognize whether a process uses TEE. We regard this as a next research topic for the scheduler to collaborate with TEE.

TrustZone showed more unusual behavior. The TA sometimes did not follow the normal application, and thus, it ran on a different core from the normal application. We imagine that OP-TEE changes the core when it accepts `SMC` instruction. This does not violate the rule of the trusted OS, but we could not determine why, leaving this question to future research.

2) CPU LOAD

The `htop` showed the core load changed from 100% to 0% sometimes on RISC-V; this was unusual because the TA should consume the CPU until the finish, and the results of `TS-perf` did not show a large difference. We analyzed the code of the `eyrie` runtime, and the unusual behavior led us to find a bug. The bug is the treatment of `handle_timer_interrupt`, which shares the platform-level interrupt controller (LPIC) between Linux and `eyrie`. The bug omits the CPU load on a core. We posit that this result was caused by a design mismatch between TEE and REE and subsequently discuss this topic in section VI-C.

VI. DISCUSSIONS

A. APPLYING `TS-PERF` TO ANOTHER TEE ARCHITECTURE

TEE implementation is not limited to core sharing, e.g., Apple iPhone's Secure Enclave [46], [47]. The secure enclave is implemented on another CPU and does not cause core change. The performance information is also hidden from the normal OS. This style of TEE architecture is not related to core change and does not affect application performance in REE. In addition, the separated-CPU TEE can avoid microarchitectural vulnerability (e.g., Spectre [48]. The vulnerability also infects the TEE (e.g., ForeShadow [49])). However, the implementation results in a higher cost. Even if the core-sharing style TEE is used, hyperthreading technology causes vulnerabilities for side channel attacks. Disabling hyperthreading is recommended for some CPUs. Fortunately, the CPU used in this paper has no hyperthreading.

The portability of `TS-perf` can be reserved on separated-CPU TEE if the time measurement works and communication between REE and TEE is guaranteed. This extension is enabled by the compiler-based performance measurement method.

On the other hand, some core-shared TEE has another cryptographic accelerator, e.g., Secure Element (SE) [50] or Rambus CryptoManager [51], which work as a root of trust [52]. GlobalPlatform defines the API from core-shared TEE to SE [53]. This style hides the performance of the cryptographic accelerator, and current `TS-perf` cannot cover the performance measurement.

B. COVERAGE OF `TS-PERF`

Fair performance comparison is a fundamental issue because current hardware and OS have performance hiding mechanisms, e.g., cache hierarchy, branch prediction, and Linux's page cache for I/O. As mentioned in section V-B, the performance degradation caused by memory encryption was not easy to disclose. These performance hiding mechanisms are effective for small access sizes and fixed patterns. In general, traditional TAs have been used for cryptographic processing, and the binaries were small, which can yield the effect of performance hiding mechanisms. However, current TAs are used by machine learning, genome analysis, privacy processing, etc. The codes and data are large, and the processing shows native performance. Since performance tuning becomes more important, `TS-perf` aims in the development of these TAs.

`TS-Perf` is not limited to the same benchmark library and can measure the precise performance of different binaries using the hardware timestamp counter. For example, `TS-Perf` can measure the binary that is optimized for REE or TEE. The results may show another perspective on this difference. This topic is the subject of our future work.

C. INTEGRATED DESIGN BETWEEN REE AND TEE

As mentioned in section I, the programming and execution environments are different between REE and TEE, which includes hardware architecture as well as software architecture. This style was effective on smartphones because the target applications are limited (e.g., key management, DRM management). However, TEE has become popular, and many normal applications want to be executed in TEE (e.g., machine learning and genome analysis). They require the execution of the same normal program in TEE.

To run normal applications without customization, SGX-LKL [54] and SCONE [55] have been developed; however, they cannot offer complete compatibility. For example, SGX-LKL does not support `fork()`. Current SCONE supports `fork()` but recommends avoiding `fork()` based on the performance problem.

We think that these problems are caused by the unfixed abstraction of TEE. As mentioned in section V-C, the mismatch between TEE and REE causes some unusual behavior. A seamless programming style in REE and TEE is desired, but the abstraction model and its support formal verification tools are not established. Hence, TEE remains in use in many research fields. `TS-perf` is a compiler-based performance measurement method and offers a seamless programming tool that can bridge REE and TEE.

VII. CONCLUSION

`TS-perf` is a general compiler-based performance measurement method and can be applied in many TEE implementations. `TS-perf` is based on the timestamp counter that is available in REE and TEE on three architectures (Arm Cortex-A, Intel x86-64, and RISC-V U540), and this method enables a fair comparison between REE and TEE. To conduct a fair comparison, we also propose to utilize the separate compilation and enable the use of the same binary in REE and TEE. The performance results showed the sameness (arithmetic performance) and difference (memory encryption and storage) between REE and TEE. The TEE results were also compared with the view from REE, and strange core change and an interrupt-handler bug were found.

REFERENCES

[1] S. Dambra, L. Bilge, and D. Balzarotti, "SoK: Cyber insurance—Technical challenges and a system security roadmap," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 293–309.

[2] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2008, pp. 315–317.

[3] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A comprehensive survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 1–36, Feb. 2019.

[4] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 18–20.

[5] Intel. *Intel Software Guard Extensions (Intel SGX) Developer Guide*. Accessed: Sep. 25, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-intel-sgx-developer-guide.html

[6] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[7] V. Costan, I. Lebedev, and S. Devadas, "Secure processors—Part I: Background, taxonomy for secure enclaves and Intel SGX architecture," *Found. Trends Electron. Des. Autom.*, vol. 11, nos. 1–2, pp. 1–248, 2017.

[8] V. Costan, I. Lebedev, and S. Devadas, "Secure processors—Part II: Intel SGX security analysis and MIT sanctum architecture," *Found. Trends Electron. Des. Autom.*, vol. 11, no. 3, pp. 249–361, 2017.

[9] R. Buhren, C. Werling, and J.-P. Seifert, "Insecure until proven updated: Analyzing AMD SEV's remote attestation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1087–1099.

[10] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Secur. Symp. (USENIX Sec)*, 2016, pp. 857–874.

[11] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure enclaves in a speculative out-of-order processor," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 42–56.

[12] (2018). *HexFive*. [Online]. Available: https://hex-five.com/

[13] 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 *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2019, pp. 1–16.

[14] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, "Keystone: An open framework for architecting TEEs," 2019, *arXiv:1907.10119*. [Online]. Available: http://arxiv.org/abs/1907.10119

[15] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.

[16] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment," 2020, *arXiv:2009.05262*. [Online]. Available: http://arxiv.org/abs/2009.05262

[17] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with CUstomizable and resilient enclaves," in *Proc. USENIX Secur. Symp. (USENIX Sec)*, 2021, pp. 1073–1090.

[18] D. Oliveira, T. Gomes, and S. Pinto, "UTango: An open-source TEE for the Internet of Things," 2021, *arXiv:2102.03625*. [Online]. Available: http://arxiv.org/abs/2102.03625

[19] Samsung. *Samsung KNOX*. [Online]. Available: https://www.samsungknox.com/en

[20] U. Kanonov and A. Wool, "Secure containers in Android: The Samsung KNOX case study," in *Proc. 6th Workshop Secur. Privacy Smartphones Mobile Devices*, Oct. 2016, pp. 3–12.

[21] M. Dorjmyagmar, M. Kim, and H. Kim, "Security analysis of Samsung Knox," in *Proc. 19th Int. Conf. Adv. Commun. Technol. (ICACT)*, Feb. 2017, pp. 550–553.

[22] D. Shen, "Exploiting TrustZone on Android," *Black Hat USA*, Aug. 2015.

[23] D. Rosenberg, "QSEE TrustZone kernel integer overflow vulnerability," *Black Hat USA*, Aug. 2014.

[24] K. Ryan, "Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 181–194.

[25] *Keystone-seL4*. Accessed: Sep. 25, 2021. [Online]. Available: https://github.com/keystone-enclave/keystone-sel4

[26] A. T. Gjerdrum, R. Pettersen, H. D. Johansen, and D. Johansen, "Performance of trusted computing in cloud infrastructures with Intel SGX," in *Proc. 7th Int. Conf. Cloud Comput. Services Sci.*, 2017, pp. 668–675.

[27] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "Sgx-perf: A performance analysis tool for Intel SGX enclaves," in *Proc. 19th Int. Middleware Conf.*, Nov. 2018, pp. 201–213.

[28] M. Bailleu, D. Dragoti, P. Bhatotia, and C. Fetzer, "TEE-perf: A profiler for trusted execution environments," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2019, pp. 414–421.

[29] R. Krahn, D. Dragoti, F. Gregor, D. L. Quoc, V. Schiavoni, P. Felber, C. Souza, A. Brito, and C. Fetzer, "TEEMon: A continuous performance monitoring framework for TEEs," in *Proc. 21st Int. Middleware Conf.*, Dec. 2020, pp. 178–192.

[30] Linaro. *Gprof in OP-TEE Documentation*. Accessed: Sep. 25, 2021. [Online]. Available: https://optee.readthedocs.io/en/latest/debug/gprof.html

[31] I. Opaniuk and J. Forissier, "Benchmark and profiling in OP-TEE," *Linaro Connect Budapest*, Mar. 2017.

[32] J. Amacher and V. Schiavoni, "On the performance of ARM TrustZone," in *Proc. IFIP Int. Conf. Distrib. Appl. Interoperable Syst.* Denmark: Springer, 2019, pp. 133–151.

[33] J. R. Blackbelt. (2013). *Processor Tracing*. Accessed: Sep. 25, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html

[34] Arm. (2010) *Coresight Trace Memory Controller Technical Reference Manual*. Accessed: Sep. 25, 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0461/b/

[35] *GNU Compiler Collection (GCC)*. Accessed: Sep. 25, 2021. [Online]. Available: https://gcc.gnu.org/

[36] *OP-TEE.Org. OP-TEE*. Accessed: Sep. 25, 2021. [Online]. Available: https://github.com/op-tee/

[37] *GlobalPlatform*. Accessed: Sep. 25, 2021. [Online]. Available: https://globalplatform.org

[38] *GlobalPlatform API Archives*. Accessed: Sep. 25, 2021. [Online]. Available: https://globalplatform.org/specs-library/

[39] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1741–1758.

[40] D. Lee. *Keystone SDK*. Accessed: Sep. 25, 2021. [Online]. Available: https://github.com/keystone-enclave/keystone-sdk

[41] K. Suzaki, K. Nakajima, T. Oi, and A. Tsukamoto, "Library implementation and performance analysis of GlobalPlatform TEE internal API for Intel SGX and RISC-V keystone," in *Proc. IEEE 19th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2020, pp. 1200–1208.

[42] (2013). *Stress-NG*. Accessed: Sep. 25, 2021. [Online]. Available: https://kernel.ubuntu.com/~cking/stress-ng/

[43] *Raspberry Pi Foundation. Raspberry Pi 3 Model B+*. Accessed: Sep. 25, 2021. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

[44] Intel. *NUC PJYH*. Accessed: Sep. 25, 2021. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/126137/intel-nuc-kit-nuc7pjyh.html

[45] SiFive. *SiFive Unleashed Board*. Accessed: Sep. 25, 2021. [Online]. Available: https://www.sifive.com/boards/hifive-unleashed

[46] Apple. *Secure Enclave Overview*. Accessed: Sep. 25, 2021. [Online]. Available: https://support.apple.com/en-am/guide/security/sec59b0b31ff/web

[47] T. Mandt, M. Solnik, and D. Wang, "Demystifying the secure enclave processor," *Black Hat USA*, Aug. 2016.

[48] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1–19.

[49] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proc. USENIX Secur. Symp. (USENIX Sec)*, 2018, pp. 1–18.

[50] GlobalPlatform. (2018). *Introduction to Secure Elements*. [Online]. Available: https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Secure-Element-15May2018.pdf

[51] Rambus. *CryptoManager Trusted Provisioning Services*. Accessed: Sep. 25, 2021. [Online]. Available: https://www.rambus.com/security/provisioning-and-key-management/cryptomanager-trusted-provisioning-services/

[52] S. Marisetty. (2017). *Demystifying Security Root of Trust*. Linaro Connect SFO. [Online]. Available: https://www2.slideshare.net/linaroorg/sfo17-304-demystifying-ro-tfinallc-83555369

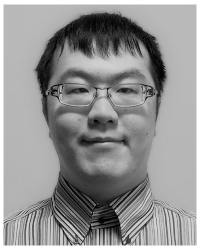
[53] GlobalPlatform. (2013). *TEE Secure Element API Version 1.0*. [Online]. Available: https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_ SE_API_v1.0.pdf

[54] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "SGX-LKL: Securing the host OS interface for trusted execution," 2019, *arXiv:1908.11143*. [Online]. Available: http://arxiv.org/abs/1908.11143

[55] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, and D. Goltzsche, "SCONE: Secure Linux containers with Intel SGX," in *Proc. Symp. Operating Syst. Design Implement. (OSDI)*, 2016, pp. 689–703.

KUNIYASU SUZAKI (Member, IEEE) received the B.E. and M.E. degrees in computer science from Tokyo University of Agriculture and Technology, and the Ph.D. degree in computer science from The University of Tokyo, Tokyo, Japan. He is currently a Senior Researcher with the National Institute of Advanced Industrial Science and Technology (AIST) and a Researcher with the Technology Research Association of Secure IoT Edge Application Based on RISC-V Open Architecture (TRASIO). His research interests include security on CPU, operating systems, and hypervisor.

KENTA NAKAJIMA received the M.S. degree in mathematical informatics from The University of Tokyo. He is currently a Researcher with the Technology Research Association of Secure IoT Edge Application Based on RISC-V Open Architecture (TRASIO). His research interests include software engineering on operating systems, system security, and software automation. He is interested in how the Linux OS and container libraries work.



TSUKASA OI is currently a Researcher with the Technology Research Association of Secure IoT Edge Application Based on RISC-V Open Architecture (TRASIO). His research interests include security of operating systems and virtual machines.

AKIRA TSUKAMOTO received the M.S. degree in computer science from Columbia University, New York. He currently works with the National Institute of Advanced Industrial Science and Technology (AIST). He has worked on products based on Cell/B.E. and Arm. His research interests include software engineering on a networks, operating systems, and system security, and he is enthusiastic regarding any kind of technical development.