On The Performance of ARM TrustZone
(Practical Experience Report)

Julien Amacher and Valerio Schiavoni[0000−0003−1493−6603]

Université de Neuchâtel, Switzerland, first.last@unine.ch

Abstract. The TRUSTZONE technology, available in the vast majority of recent ARM processors, allows the execution of code inside a so-called *secure world*. It effectively provides hardware-isolated areas of the processor for sensitive data and code, *i.e.*, a trusted execution environment (*TEE*). The OP-TEE framework provides a collection of toolchain, open-source libraries and secure kernel specifically geared to develop applications for TRUSTZONE. This paper presents an in-depth performance- and energy-wise study of TRUSTZONE using the OP-TEE framework, including secure storage and the cost of switching between secure and unsecure worlds, using emulated and hardware measurements.

Keywords: Trusted Execution Environment · ARM · TrustZone · benchmarks

1 Introduction

Internet of Things (IoT) devices are expected to offer the pervasive computing that was promised at its advent [47]. The economic impact of the IoT ecosystem has created many new business opportunities and is expected to continue growing rapidly. As a result, the number of devices owned per user is anticipated to increase up to 26 by 2020 [44]. ARM, expects 275bn active devices by 2025 - a 11× improvement over 2019 [6] - while already having sold 100bn processors. For instance, Figure 1 reports the sales for ARM processors in the last 20 years.

These IoT devices gather, distribute and process information on their own, effectively pushing intelligence to edge devices. Due to their nature, these devices are mostly nomad: easy to relocate, designed as wearable, embedded in vehicles or left in remote locations. As such, assets need to be protected from attackers, in particular those easily subject to physical tampering. Hence, ensuring that confidential data is processed in a secure manner, even in hostile environments, remains a challenging prerequisite for such devices. Indeed, an attacker with physical access can relatively easily inspect and modify the execution workflow of any program. Nowadays, even more disturbing attacks not requiring physical access are surfacing [51], reinforcing the need to exploit hardware-based security mechanisms when available. Hardware-based protections offer an additional security layer, by physically separating processing of secure and non-secure data components. These can be dedicated processing chips (hardware security modules –HSM–), or regular chips to which security extensions were added. Examples of the latter include Intel's *Software Guard Extensions* (*i.e.*, SGX [21]) since the Skylake architecture (2015), or ARM's TRUSTZONE [7] since ARMv6 (2008).

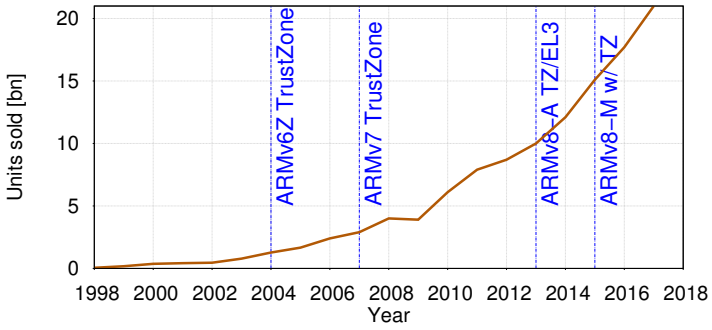


Fig. 1: Sales and popularity of ARM processors in the last 20 years [5,4]

ARM devices are often battery-powered and must therefore make optimal use of their limited energy capacity. This is especially true nowadays, when battery capacity is becoming the limiting factor when deploying new functionalities. Despite the availability of such devices on the market, to the best of our knowledge we could not find a public study on the performance and energy-related consumption for these security extensions.

The contributions of this work are as follows. We begin by providing the first public experimental analysis of the performance and energy requirements of the TRUSTZONE security extensions based on hands-on metrics. Second, we report on the advantages and limitations of OP-TEE [26], an open-source framework that supports TRUSTZONE. Third, we provide a methodology to extend the kernel of OP-TEE in order to offer new syscalls inside TRUSTZONE. We leverage this methodology to implement two new additional syscalls, *e.g.*, to fetch thermal metrics and for secure time measurements in the TRUSTZONE. Finally, we report on our in-depth experimental analysis along several dimensions (including energy) of the current secure processing capabilities offered by some widely popular IoT devices (*i.e.*, Raspberry Pi) shipping TRUSTZONE processors. Our results are put into perspective by comparing them against an emulated environment aware of the TRUSTZONE extensions.

The paper is organized as follows. §2 describes the TRUSTZONE architecture and key concepts of world isolation. §3 explains how the kernel was extended to expose new syscalls within TRUSTZONE, how all the data was gathered, as well as the hardware and software tools that were developed. §4 presents our in-depth evaluation using real hardware and under emulation, for several hardware components (*e.g.* CPU, memory, secure storage) and metrics (*e.g.* performance, energy and power consumption). We discuss some lessons learned in §5, before concluding in §6.

2 Background

This section provides some background on TRUSTZONE. First we define a few terms used throughout this paper. §2.1 describes TRUSTZONE's main mechanisms and limitations, while §2.2 introduces OP-TEE.

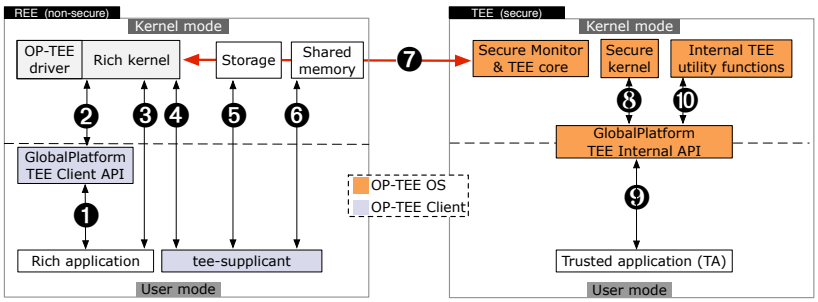


Fig. 2: TRUSTZONE components and interaction workflow.

Rich Execution Environment. The REE (or *normal world*) is the regular, non-secure operating system of a device. The memory, registers, and caches are not isolated or protected by any hardware mechanism. Typically, the REE is not focused on security and is difficult to review for security vulnerabilities, due to its large size and complexity.

Trusted Execution Environments. Also called TEE or *secure OS*, it is the so-called *secure world* operating system part of the TRUSTZONE specifications. It complies with the GlobalPlatform's TEE System Architecture specifications [57], a set of operations offered to secure applications. These include interactions with persistent (secure) storage [57, Chapter 5], memory [57, Chapter 4.11], and cryptographic operations [57, Chapter 6]. As such, a secure application can easily be ported to another platform, due to the standardized nature of available services. Similar to what a non-secure operating system offers to its running applications, the TEE offers access to special services only available to secure applications (such as the secure storage feature, which we evaluate). This environment has a small footprint, contrary to a full-fledged operating system, and only implements the very minimal set of features required to operate. Its small size makes it simpler to review for security vulnerabilities, as any could potentially compromise all secure applications.

Trusted Application. A trusted application (*TA*), also called secure application is designed to be run exclusively inside the secure world. It uses services provided by the TEE kernel to access resources, specifically: (1) disk via the secure storage subsystem exclusively, (2) TCP/IP sockets, (3) memory allocation, (4) other custom services. Trusted applications provide services to either standard userland programs or other TAs. OP-TEE expects TAs to be written in C.

2.1 TRUSTZONE in a nutshell

This section describes the main components of the TRUSTZONE architecture, also depicted in Figure 2 alongside their interfaces.

Overview. TRUSTZONE is a hardware feature implemented in recent ARM processors. It enables physical separation of different execution environments, namely TEE and REE. Its working principle is very similar to a hypervisor, the main difference being that no emulation is performed and that all isolation is offered at the hardware level. Both secure (TEE) and normal worlds (REE) share the underlying physical processor. The secure world has unrestricted access to memory regions, hardware and devices.

This is realized by using an additional addressing line, the NS (Non Secure) bit. Hardware checks performed by the TZASC (TRUSTZONE Address Space Controller) [42,50] determines, if the access is authorized based on this NS-bit.

Memory. Parts of the memory can be isolated for exclusive use by the secure world by means of special hardware support. The memory management unit (MMU) is secure-world aware, and secure and non-secure descriptors are stored alongside each other. The differentiation is done by the *Non-secure TLB ID* (NSTID) [12], an extra bit of the TLB. The secure applications (TAs) must fit in the on-chip memory. Due to high costs of the secure memory, it is usually limited in size, in the order of 3-5MB. Hence, TAs are expected to have small memory footprints and only contain the minimal subset of features required. Clearly, this reduces the attack surface exposed by TAs.

Interrupts. The *Fast Interrupt* (FIQ) secure interrupt mode is used exclusively by devices residing in a memory region allocated to the secure world. As such, regular interrupts (IRQ), which are of lower priority, cannot be used to prevent the secure world from executing, in particular if a physical secure clock (*i.e.*, RTC) is used. Secure clocks are crucial to ensure a TA is safely executed: an external clock is a common attack vector and can be easily tampered with [53]. Latest ARM processors include secure clocks.

World Switching. Switching between worlds requires the state of the processor to be saved and then restored, respectively when entering and exiting a new world. Processor registers are saved by the monitor when entering, and restored when leaving the secure world. The NS-bit is changed accordingly. Normal world applications use TRUSTZONE indirectly, by invoking functionalities implemented in a dedicated TA. When in PL-1 [43,1] privilege level, a special hardware instruction, *Secure Monitor Call* (SMC), allows switching between worlds. Recent Cortex-A processors [48] support SMC calls by the kernel in the normal world. Entry to a different world (from secure to unsecure and vice versa) is done on a core-basis, thus limiting the parallel execution of TAs to the number of available cores. To enter the secure world, a kernel thread executes the monitor, which in turn issues the SMC instruction to the CPU [8,29]. Calls to SMC by a processor not in kernel mode trigger an undefined exception trap. TAs can be called from userland programs residing in the REE or from other TAs. The latter is particularly useful to reduce code duplication and to keep the TA's attack surface minimal. Data is passed back and forth between worlds by memory pointers or direct copies.

Secure storage. TRUSTZONE supports persistent data storage for TAs using secure storage. Objects are stored encrypted on disk, and are signed for anti-tampering countermeasure. TAs access the files in cleartext: the TEE layer runs the cryptographic stack transparently. These files have a unique numeric name based on a counter. An encrypted index of files is maintained alongside the files. Operations on the index are atomic, ensuring integrity protection by means of a hash tree data structure that guards the index. To protect against storage replay attacks, an eMMC storage device (*embedded MultiMediaCard*, a type of non-volatile, non-removable solid-state storage device [22]) is required. This security feature is entirely implemented in the eMMC storage in the form of *Replay Protected Memory Block* (RPMB) [55].

Key Management. The key manager starts with a device-specific key, the *Secure Storage Key* (SSK). It is derived from two pieces of information unique to each device's processor: the chip identifier and the hardware key. The *TA Storage Key* (TSK) is a per-

Framework	License	Technology
OP-TEE [26]	BSD	TRUSTZONE
Trustonic TEE [38]	Commercial	TRUSTZONE
Open TEE [52]	Apache License 2.0	TRUSTZONE
OpenEnclaves [23]	MIT	SGX1 & TRUSTZONE
TLK [54]	BSD	NVIDIA Tegra
Android Trusty TEE [2]	Apache License 2.0	TRUSTZONE[1]

[1]: emulated under Intel's VT
Table 1: Existing frameworks for TEE-based applications.

TA key, derived from the SSK and the TA's UUID identifier. The *File Encryption Key* (FEK) is a per-file key generated upon file creation. It is used to protect the file contents, including its metadata, and is encrypted using the TSK.

Resilience to attacks. It is of paramount importance to ensure that only trustworthy applications are deployed to the secure world. Vulnerabilities in any TA, the TEE or a compromised secure kernel do compromise the security of the secure world. Prevention against buffer overflow attacks in the secure world are currently only provided using basic stack canaries [31]. Future support for ASLR (Address Space Layout Randomization) will improve resilience against those attacks. Finally, there exist mitigations against Meltdown and Spectre speculative execution attacks [15,13,14,16]. Covert data channels [45] can also be used when required.

2.2 The OP-TEE Trusted OS

While there are few options (Table 1) to develop applications for TEEs, we rely on OP-TEE, due to its fast development cycle and native support for the TRUSTZONE.

OP-TEE is a security framework that includes several components: a minimal secure-world operating system (the OP-TEE OS [26]); the *tee-supplicant* [30], offering normal world services to the secure world; a complete build toolchain [24], the testing tool [28] (*OPTEE sanity testsuite*), a secure privileged layer enabling world switching, a basic REE image, and several utility functions for developers to implement TAs. OP-TEE is flexible and can be deployed to platforms for which there exists a manifest, that lists the dependencies required to build for the platform it describes, as well as its hardware characteristics. Additionally, the Qemu open source emulator [33] allows to deploy and evaluate OP-TEE in emulated mode on ubiquitous machines. The TEE interface implemented in OP-TEE is compliant with the GlobalPlatform's specifications.

Details. OP-TEE imposes a specific interface regarding TA interactions initiated from the REE. First, a request to load the desired TA is made by passing its UUID to *TEEC_InitializeContext* which returns a context object. The UUID is defined at compile-time and must be unique amongst all TAs. Next, this context is passed to *TEEC_OpenSession* which returns a session. This session is then used to invoke actual services in the TA using the *TEEC_InvokeCommand*, which takes as parameters the service identifier as well as any optional parameters. A single session can be used to call *TEEC_InvokeCommand* any number of times. Sessions are finally closed using *TEEC_CloseSession* and ultimately, the context is closed by calling *TEEC_FinalizeContext*.

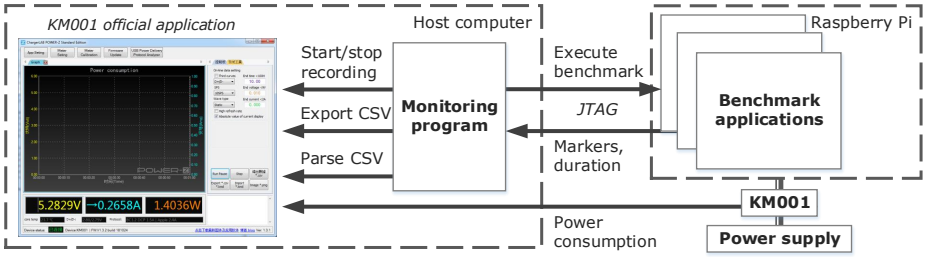


Fig. 3: Experimental setup and approach used to run our measurements

To support multiple sessions, the TA must be compiled with the *TA_FLAG_MULTI_SESSION* flag set. OP-TEE signs TAs with a private RSA key, but the toolchain does not allow a unique key per-TA (all TAs are signed with the same device key). Upon TA loading, the OP-TEE core checks the integrity of the TA by verifying its signature based on its signed header. The framework includes a minimal OS that offers services to TAs, and leverages the tee-supplicant application to access resources residing in user land.

3 Methodology

This section describes the tools and techniques used to carry out our evaluation. We focus on four metrics : (1) execution time for various types of benchmarks (CPU-bound, volatile and non-volatile memory), (2) power consumption under different CPU governors, (3) energy consumption, and (4) thermal behaviour of the CPU.

Hardware Measurement Tools. Energy and power measurements are carried out using a Power-Z KM001 unit [32], plugged in-between the USB power supply and the Raspberry Pi device. The variant used in our testbed features two main USB ports (to provide power and one from where the power is drawn) of the current mainstream USB types (type A, micro and type C). In our configuration, type A is used for both input and output of power delivery. An additional (micro) USB port is used to fetch power consumption measurements. The KM001 unit supports different USB protocols, including USB PD (Power Delivery) 2.0 and Qualcomm QC (QuickCharge) from version 2.0 up to 4.0. This configuration allows the power used by the Raspberry Pi to be measured directly as the losses of the power supply itself are not taken into account. We use this device to measure only power [W] and energy [Wh], for which it produces 1 record per second. Unfortunately, the software (Figure 3, left) provided by the unit manufacturer is a closed-source 32-bit Windows binary, and the protocol used to exchange messages over USB is undocumented. To overcome these limitations, we used the following approach. Specific markers (*e.g. start recording* and *stop recording*) are generated during execution of benchmark applications, allowing for precise recording of areas of interest (Figure 4). These markers are monitored by a custom program (on a separate node) that pilot the Windows binary (Figure 5). The pilot sends automated messages to the binary instance using the Win32 API through P/Invoke (Platform Invokation Service) [11] issued by a monitoring program implemented in C#.

CPU Governors. The Linux kernel supports several CPU governors [46], used to adjust the frequency of each core depending on its load and temperature. Several options

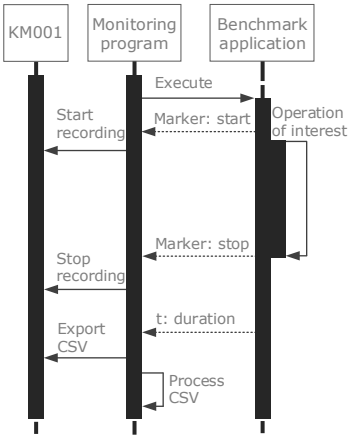


Fig. 4: Use of markers

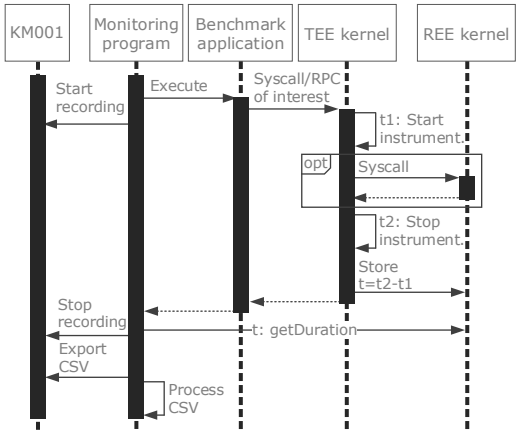


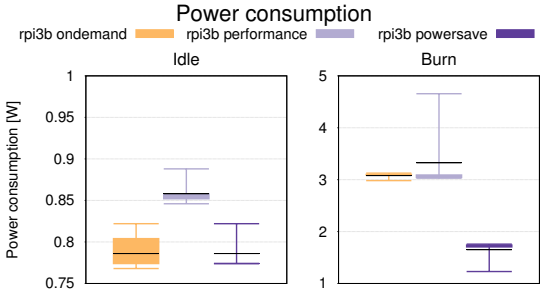
Fig. 5: Microbenchmarking: workflow

exist: `powersave` and `performance` for minimum and maximum operating frequency; `ondemand` toggles between the previous two, and a more `conservative` mode that operates less aggressively; `userspace`, to manually set the CPU frequency; and `schedutil`, where the frequency is set by the scheduler. The core frequency is increased during the execution of stressful workloads and reduced right after, for instance when the maximum temperature is reached in order to prevent overheating. This is different from a hardware thermal throttling, which tries to prevent damage caused by excessive heat. The OP-TEE kernel uses `powersave` governor by default. This reduces heat output by reducing the frequency of the core clocks, allowing passive cooling - even without heatsink - but also negatively impacts performance. In a compute-intensive datacenter, one would typically use the `performance` governor. Instead, if energy constraints are important, the `powersave` mode is best suited. Our benchmarks consider both governors and compare them for REE and TEE executions.

Timing issues. Initially, we planned on porting STRESS-NG [36] to run inside TRUSTZONE. Unfortunately this proved to be not straightforward, given its reliance on system calls not available inside the TEE kernel. As such, we decided to implement custom ad-hoc benchmark applications. Execution time is measured using either the `gettimeofday(2)` [18] or the `clock_gettime(3)` [10] syscall, which support the following parameters:

1. `CLOCK_REALTIME`: the realtime clock of the system, can be adjusted by NTP and thus can go forward and backwards.
2. `CLOCK_MONOTONIC`: a monotonic time since an unspecified starting point (usually system startup, as is the case with our setup)
3. `CLOCK_PROCESS_CPUTIME_ID`: per-process timer
4. `CLOCK_THREAD_CPUTIME_ID`: thread-specific CPU-time clock

For our experiments we exclusively use `CLOCK_MONOTONIC`. Our benchmarks include the instrumentation delay, *e.g.*, the overhead introduced by the measurement itself. This is especially important from the TEE perspective (*i.e.*, inside a TA) where one syscall can lead to a second one if REE needs to be accessed (*e.g.*, Figure 2-❾ and Figure 2–❼).



Governor	Idle		Burn	
	W	BTU/h	W	BTU/h
ondemand	0.78	2.66	3.08	10.51
performance	0.86	2.93	3.32	11.33
powersave	0.78	2.66	1.65	5.63

Table 2: Average power consumptions for idle and burn experiments (see Figure 6)

Fig. 6: Idle (left) and burn (right) power consumption.

Kernel and OP-TEE modifications. To access and store the monotonic time and temperature from within a TA using the secure kernel, and to retrieve it later on within the REE, we extended the kernel with four new system calls: `TEE_GetCpuTemperature`, `sys_ktraceadd`, `sys_ktraceget` and `sys_ktracereset`.

To gather the temperature measurements, we used two methods: (1) software, via thermal APIs[1] and (2) external hardware sensor. Originally, we planned on using a script to record the temperature at fixed intervals during the CPU stress tests executed by userland threads. However, since kernel threads executing the TAs have a higher priority, the userland threads were starved and thus did not produce enough data points. This is a typical scenario of normal world starvation occurring when TAs monopolize all cores. We overcome this problem by accessing the CPU temperature from inside the TA, and sending it periodically to the monitoring software for safekeeping. To use the temperature gathering syscall from within the TA, we additionally had to implement the corresponding TEE kernel syscall wrapper. An extensive walkthrough on this process is given in Appendix A.

4 Evaluation

This section presents our in-depth evaluation and performance analysis, the main contribution of this work. Energy results are always presented by systematically excluding idle energy consumption, *e.g.*, we only show the energy cost of the given operation. Energy requirements are shown on a per-operation fashion. To prevent thermal throttling, all tests run while the onboard chip is actively cooled.

Evaluation Settings. We use the Raspberry Pi 3B, a popular yet representative single-board device, equipped with Broadcom BCM2837 *System-On-Chip* (1GB of RAM, ARM Cortex A53 quad core running at 1.2GHz). For some of our measurements, we compared the hardware experiments against a modified version of the Qemu emulator provided by OP-TEE with support for TRUSTZONE [34]. This mimics the scenario of an Infrastructure-as-a-Service provider offering access to ARM nodes (as virtual machines) to cloud tenants without having the corresponding hardware infrastructure and thus relying on TRUSTZONE virtualization [49]. Qemu uses the Cortex A53 emulation

[1] `/sys/class/thermal/thermal_zone[0-9]+/temp`

profile on an Ubuntu host residing on a VMWare ESXi [40] machine equipped with an i7 6820HQ running at 2.7GHz. Note that the Raspberry Pi 3B lacks support for secure boot and hardware separation of memory and peripherals [27], hence these aspects of the TRUSTZONE ecosystem could not be evaluated and are left for future work. Finally, we do not override the default secure storage key (SSK) provided by OP-TEE.

Power consumption. We start by measuring the idle and under-stress (*burn*) power consumption of our hardware unit. We evaluate how the three different CPU governors (`ondemand`, `performance`, and `powersave`) behave. The idle measurements use the standard REE kernel image provided by OP-TEE, without any user-intensive applications nor TAs running. Burn measurements run the prime benchmark, a single-threaded TA which computes the first 20000 prime numbers before exiting. We run 8 instances in parallel, ensuring maximum heat output on the 4 cores. Measurements start 60 seconds after the benchmark instances. Figure 6 shows our results, respectively for idle (left) and burn (right) experiments. Table 2 shows the average W and BTU/h. We use a box-and-whiskers plot: the first and third quartile are shown as a colored box, the median as horizontal black bar. Min/max values are also included. Results for *ondemand* and *powersave* are on par with the *ondemand* governor, in particular when the CPU frequency is set at 600MHz. As expected, we observe higher power consumption using the `performance` governor even in idle, as the cores are boosted up to 1.2GHz. Overall, the board's power consumption is very low, in particular below 1W in idle mode.

Load & unload TAs. Next, we measure the time required to load and unload a TA inside the TRUSTZONE, respectively executing *TEEC_InitializeContext* [56, Chapter 4.5.2] and *TEEC_FinalizeContext* [56, Chapter 4.5.3] functions. We compare results obtained with a TA of size smaller and another one of size larger than the 512kB L2 cache of the Broadcom BCM2837 processor, respectively 102kB and 517kB. Our experiments show no significant difference between TAs of different sizes. For each configuration, Figure 7 shows average and standard deviation over 10k executions. We include the time spent to execute an empty function inside the TA once it is loaded (1.31ms), to give a baseline of comparison.

Surprisingly, our results do not show a significant differences on subsequent loadings compared to the first loading, despite the tee-supplicant is supposed to cache the TA code. We will investigate this aspect in future work.

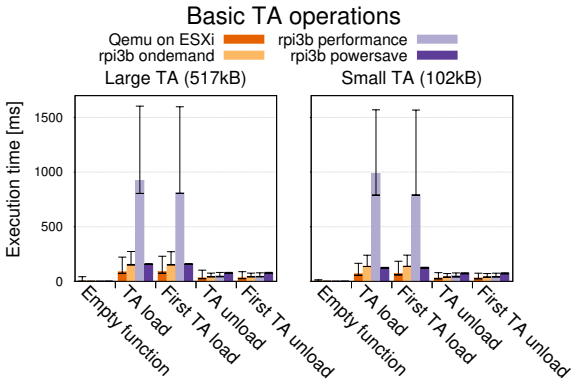


Fig. 7: Basic TA operations: loading, unloading and successive calls to load/unload the same TA.

Context (World) Switching. Switching between worlds is a key operation when deploying applications that execute inside and outside the TRUSTZONE. To measure the switching time, we implemented an ad-hoc benchmark made by a host application and a TA. Both programs record the monotonic time when entering and exiting the world

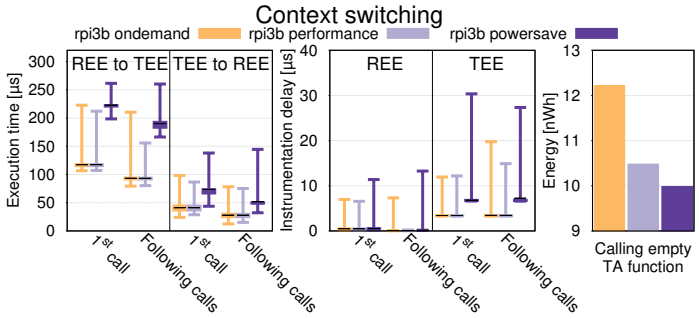


Fig. 8: World switching performance and energy requirements

in which they reside. The host issues a call to an almost empty function, which only contain time-measuring code. Two calls are made to the TA per session, recording the time taken to switch between TEE and REE, and vice versa. Figure 8 (left) shows these results. To evaluate possible caching effects, we also include the results obtained for all the calls following the first one. As expected, it is more time-consuming to switch from the REE to the TEE (110µs with the performance-oriented governors) than the opposite (47µs). The instrumentation delay (Figure 8, center) is the difference between two consecutive calls to the time measurement function. An increased instrumentation delay is observed in the TEE compared to the REE, due to the additional world switch. Finally, we also evaluate the energy spent for calling an empty TA function from the REE (Figure 8, right). The timer starts and stops when leaving and re-entering the REE, respectively. The *ondemand* governor is the most energy-eager (up to 12.1 nWh), while *powersave* is the most energy efficient.

Volatile Memory. Next, we consider simple in-memory operations (*e.g.*, read and write, sequential or at random), for two different sizes of volatile memory (1MB and 100KB) used by the REE and the TEE. We consider inter- (REE←TEE) and intra-world (*e.g.*, REE↔REE, TEE↔TEE) memory readings, as TRUSTZONE restrictions prevents reading TEE memory from the REE. We compute the average and standard deviation over 100 run, always using the high-resolution monotonic counter. Figure 9 shows our results, for the Raspberry Pi device with 3 CPU governors and using Qemu. Performance of accessing a single byte in TEE memory from the TEE is on par with accessing REE memory from the TEE, on average 0.01µs, around 2× under emulation. Interestingly, using memory from within the TEE is also less energy eager (Figure 10), also verified by the cost of the single operations in the various configurations. We observe how the operations in the TEE↔TEE case are on average 2× faster on bare metal and 1.2× under emulation than in the other cases.

Secure Storage: performance. We evaluate the performance of TRUSTZONE's secure storage via the corresponding GlobalPlatform's API implemented by OP-TEE. Specifically, we benchmark the cost of creating, writing, reading and closing objects inside the secure storage area, for two different object sizes (100KB and 1MB), although current memory allocator limitations prevented to cover some cases [35,19,20,39]. Figure 11 (left) shows that closing and deleting objects are fast operations, and opening and writing are the slowest ones. Iterating over objects in the secure storage (*e.g.*, the

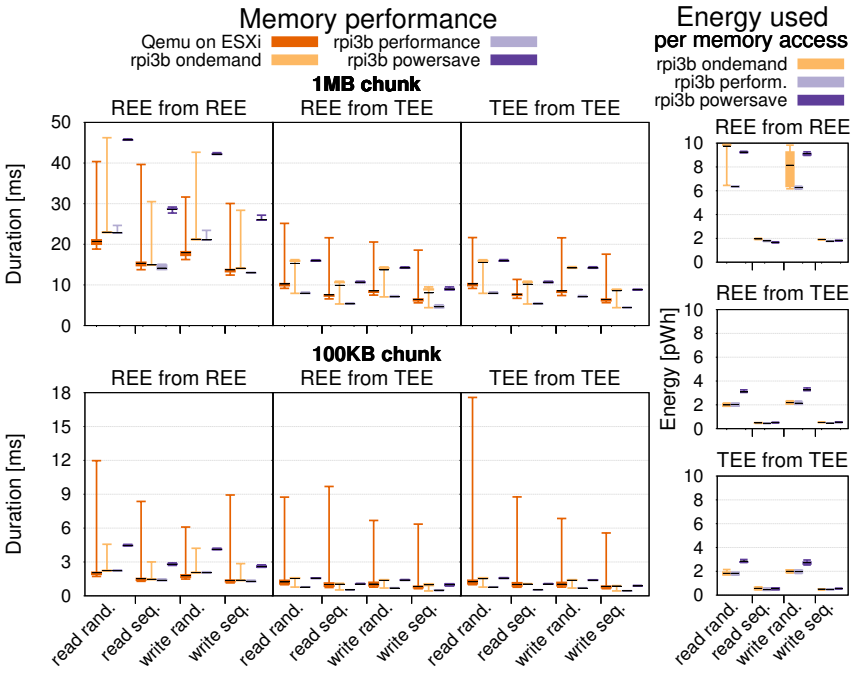


Fig. 9: Benchmark for memory ops

Fig. 10: Energy: memory accesses

execution of a `find` operation) is slow, up to a few hours in the worst case (Figure 11, right). Adding more objects in secure storage degrade the results even more (up to $2.01 \times object\_count\_ratio$).

Secure storage: cost breakdown. To understand how each low-level syscall affects the performance of a file-system inside the secure storage, we implemented a simple microbenchmark, inside `ree_fs_create` and `ree_fs_write`. Specifically, these tests create and write data into a new object. Figure 14 shows a breakdown cost using stacked bars for writing and creating files. These two functions are atomic and thus are surrounded by a monitor (mutex) which adds a considerable delay (not shown) regarding the *write* operation. The impact is negligible on the *create* operation. We observe that opening the file and setting the filename accounts for the most time spent.

Secure Storage: energy. Being a feature often used by nomad devices with low energy autonomy, we deeply investigate its energy impacts. Figure 12 shows that creating objects is the most energy-demanding (up to 403µWh), irrelevant of the size. Power consumption of writing objects is dependent on their size. Interestingly, the *ondemand* governor achieves slightly worse results when creating a file, whereas for closing and deleting files it stands out. Figure 13 shows the energy requirements to iterate over a single stored object (top) [57, Chapter 5.8] during enumeration of all stored objects in secure storage or rename (bottom) a single object, when additional 10 or 100 objects (of the same size) are already in the secure storage. We execute this test for 2 different

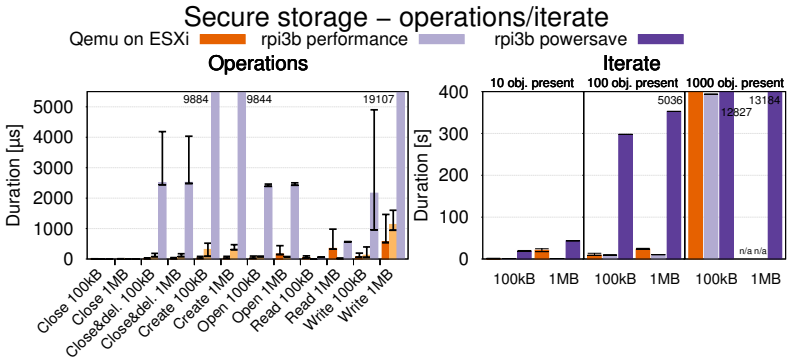


Fig. 11: Secure storage: basic operations (left) and iteration (right)

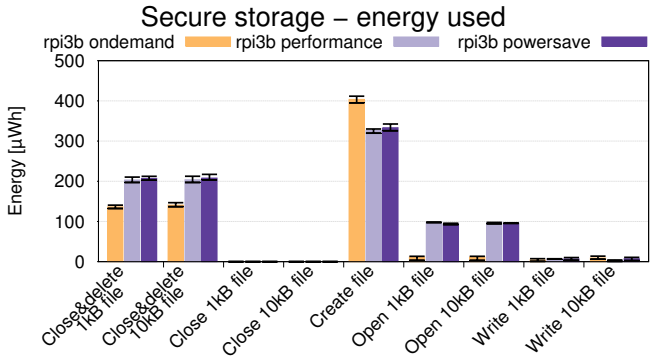


Fig. 12: Secure storage: energy measurements for basic operations

file sizes (1kB and 10kB). We observe that the energy required to iterate over a single object depends on the number of objects stored (in particular when using `performance` and `ondemand`), whereas the size of the object is irrelevant.

CPU Benchmarks. To benchmark the raw performance of the ARM processors of our units, we implemented and deployed a single-threaded TA that executes a CPU-bound task, *e.g.*, computes the first 20000 prime numbers. We run multiple instances concurrently, and while they execute we also gather energy measurements (for all cases minus the emulation mode). Figure 15 presents these results. As expected, the `performance` governor ensures the fastest computing time. Due to emulation costs, the Qemu results are the worst ones. As the number of instances exceed the available hardware cores, we observe an increase of energy consumption. Overall, in this benchmark the `ondemand` governor is the most energy eager. This can be explained by the fact that adjusting the core frequencies (from 600MHz and 1.2GHz) seems to be a relatively costly operation [41].

Thermal benchmarks. We conclude our evaluation by looking at the thermal envelope of the *SoC*. To do so, we execute 8 concurrent instances of the prime benchmark inside TRUSTZONE. Figure 16 presents the measurements fetched using the kernel's thermals API. Additionally, we monitor the surface temperature of the chip using a Texas Instruments LM35 precision linear sensor with the help of an external micro

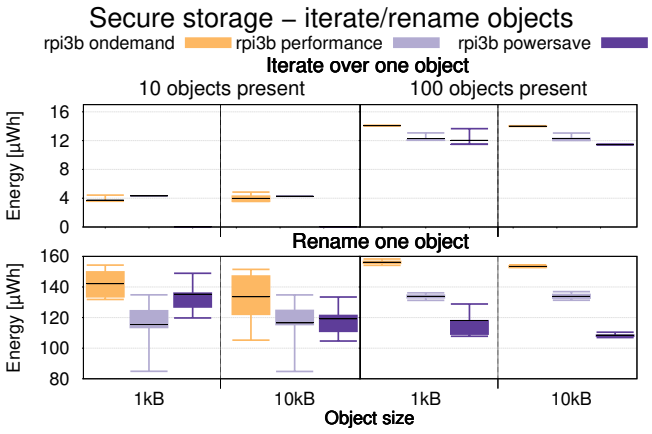


Fig. 13: Secure storage, energy to iterate (top) and rename (bottom)

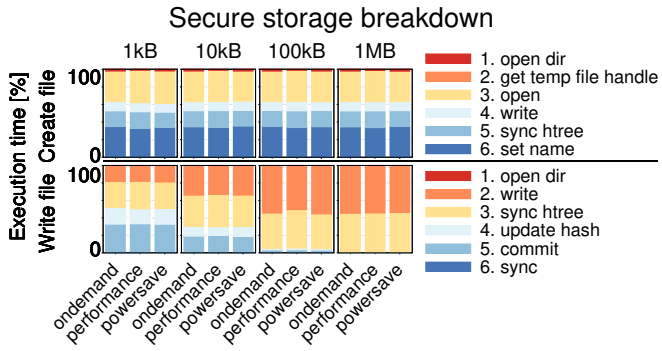


Fig. 14: Secure storage breakdown for two operations: *create* and *write*

controller. Thermal conductivity between the *SoC* and the LM35 is ensured by using a thermal compound (Arctic MX-4[3]). The ambient temperature is of around 21.9°C. Results returned by the LM35 are calibrated and checked at rest against a Fluke thermocouple, and against a Flir E4 [17] thermal camera (see pictures in Figure 17). Marked points in Figure 16 refer to measurements done using the thermal camera. We observe a small margin of error of 3°C, and a discrepancy between the thermals API and the LM35 of over 15°C at times. This could be problematic because the measured surface temperature exceeds the rated continuous temperature of 85°C specified by the chip's manufacturer. In this situation, the thermals API returns an incorrect temperature that is well below the acceptable temperature. As a consequence measures which should be taken to reduce the temperature, such as software thermal throttling, are not undertaken. A passively cooled Raspberry Pi should therefore only operate in `powersave` mode or risk being hardware throttled or worse, suffer damage. An actively cooled system on the other hand can operate in any mode and stay well within acceptable conditions, even without additional heat sink. Once the maximal temperature is reached, recovery time is around 8 minutes when passively cooled and less than a minute with active cooling.

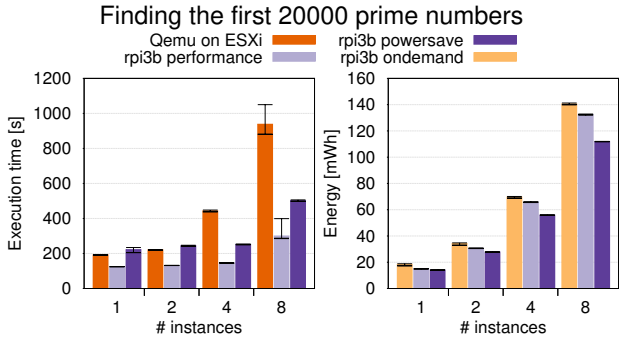


Fig. 15: CPU benchmark: processing delay and energy requirements.

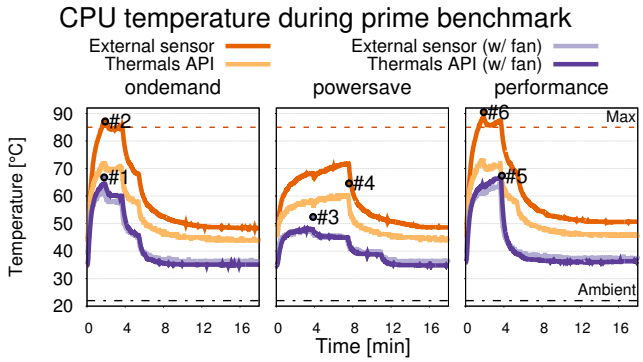


Fig. 16: Evolution of CPU temperature with different cooling modes and governors.

5 Lessons Learned

This section reports on a few lessons learned during this experimental work.

Memory limitations. By default, 32MB are dedicated to OP-TEE, of which: 1MB for TEE memory, 1MB for PUB (non-secure RAM) memory, and the remaining 30MB for TAs. Each TA has two compile-time options, *TA_STACK_SIZE* and *TA_DATA_SIZE* (in *user_ta_header_defines.h*), defining the stack size and heap size that can be utilized by a TA. These values are set at very low values by default, 2kB and 32kB respectively [25]. For larger memory allocations, the TA's MMU L1 table must be set accordingly, as the default mapping is 1MB. We were unable to allocate more than 3MB for a single TA, even with shared memory enabled. Consequently, the OP-TEE benchmark framework [9] could not be used.

Compliance to standards. The GlobalPlatform's implementation in OP-TEE is not error-free and some parts of the implementation do not comply fully with the specification. For instance, the *TEE_BigIntAdd* [57, p. 252] function, contrary to its definition, does not allow to use the same pointers for both input and output [37]. Being relatively new, OP-TEE is improving rapidly. While this offers great advantages, such as mitigations against the latest attacks, it also introduces incompatibilities by deprecating older APIs. However, the GlobalPlatform consortium offers strong incentives for TEE vendors to comply with their API, which is unlikely to introduce breaking changes.

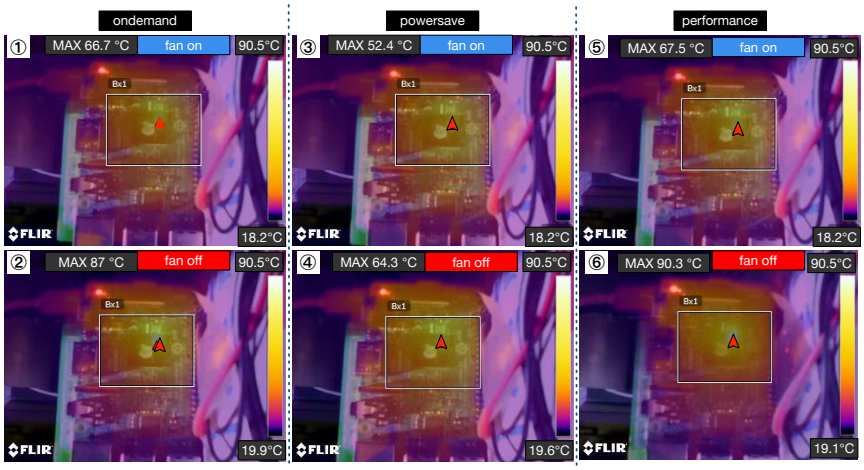


Fig. 17: Raspberry Pi thermal behaviour during processor stress benchmarks.

Establishing this level of compliance ensures interoperability of TAs between existing TEE solutions which is undeniably of great interest to secure application developers.

Developers toolchain. The OP-TEE framework groups all required dependencies in a single project while also including several components of its own, such as the secure kernel. This greatly facilitates development of secure application by reducing setup and development efforts. The OP-TEE project includes a few TA examples and host applications, which are a good foundation to introduce the TEE paradigm.

6 Conclusion

TRUSTZONE is a widely available technology that offers Trusted Execution Environment guarantees to low-energy devices. The goal of this practical experience report was to uncover the performance of these systems. To perform our experiments, we had to extend both secure and rich kernels so that secure timing measurements and thermal metrics could be fetched from within TRUSTZONE, for which we provide detailed explanations in Appendix A. Our work highlights several advantages as well as limitation of the currently available software platforms, such as the OP-TEE framework chosen in our case, to implement and deploy TAs. We would like to point out two major limitations. (1) the lack of several basic features inside the REE kernel for security reasons, which materialize in the lack of basic syscalls (*e.g.* `fopen`, `msgget`). For this reason, it is paramount to reduce syscall dependencies when developing TAs. (2), the current limitations regarding memory allocation and addressing, which could negatively affect the facility to deploy more complex TAs inside TRUSTZONE. We hope this work will provide useful insights to TRUSTZONE software developers.

Acknowledgments

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the LEGaTO Project (`legato-project.eu`), grant agreement No 780681.

References

1. AArch64 Exception Handling - System calls to EL2/EL3. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch10s02s04.html.
2. Android Trusty TEE. https://source.android.com/security/trusty.
3. Arctic MX-4. https://www.arctic.ac/ch_en/mx-4.html.
4. ARM Everywhere. https://hexus.net/static/arm-everywhere/.
5. ARM Financial Results. https://www.arm.com/company/investors/financial-results.
6. ARM Inside The Numbers - 100bn. https://community.arm.com/processors/b/blog/posts/inside-the-numbers-100-billion-arm-based-chips-1345571105.
7. ARM TrustZone Developer. https://developer.arm.com/technologies/trustzone.
8. ARM1176JZF-S Technical Reference Manual - 2.12.13. Secure Monitor Call (SMC). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/ch02s12s13.html.
9. Benchmark framework. https://github.com/OP-TEE/optee_os/blob/master/documentation/benchmark.md.
10. clock_gettime(3) - Linux man page. https://linux.die.net/man/3/clock_gettime.
11. Consuming Unmanaged DLL Functions. https://docs.microsoft.com/en-us/dotnet/framework/interop/consuming-unmanaged-dll-functions.
12. Cortex-A9 Technical Reference Manual - 6.3. Memory Access Sequence. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/Ciheiecd.html. Accessed: 2018-12-09.
13. CVE-2017-5715. https://nvd.nist.gov/vuln/detail/CVE-2017-5715.
14. CVE-2017-5753. https://nvd.nist.gov/vuln/detail/CVE-2017-5753.
15. CVE-2017-5754. https://nvd.nist.gov/vuln/detail/CVE-2017-5754.
16. CVE-2018-3639. https://nvd.nist.gov/vuln/detail/CVE-2018-3639.
17. Flir E4. https://www.flir.com/products/e4/.
18. gettimeofday(2) - Linux man page. https://linux.die.net/man/2/gettimeofday.
19. Hikey: trying to allocate more physical memory to secure world. https://github.com/OP-TEE/optee_os/issues/1396.
20. How to alloc 10M memory by TEE_Malloc(). https://github.com/OP-TEE/optee_os/issues/2090.
21. Intel SGX. https://software.intel.com/en-us/sgx.
22. Kingston Embedded Solutions. https://www.kingston.com/en/embedded/emmc.
23. Microsoft OpenEnclave Framework. https://github.com/Microsoft/openenclave.
24. OP-TEE Build on Github. https://github.com/OP-TEE/build. Accessed: 2018-12-04.
25. OP-TEE FAQ on Github. https://github.com/OP-TEE/OP-TEE_website/tree/master/faq. Accessed: 2018-12-04.
26. OP-TEE OS on Github. https://github.com/OP-TEE/optee_os. Accessed: 2018-12-04.
27. OP-TEE Raspberry 3B platform specific documentation. https://www.op-tee.org/docs/rpi3/.
28. OP-TEE sanity testsuite on Github. https://github.com/OP-TEE/optee_test. Accessed: 2018-12-04.
29. OP-TEE source. https://github.com/OP-TEE/optee_os/blob/master/core/arch/arm/kernel/generic_entry_a64.S. Accessed: 2018-12-09.
30. OP-TEE Supplicant on Github. https://github.com/OP-TEE/optee_client/tree/master/tee-supplicant. Accessed: 2018-12-04.
31. OPTEE-OS kernel thread.c init_canaries. https://github.com/OP-TEE/optee_os/blob/master/core/arch/arm/kernel/thread.c#L150.

32. POWER-Z KM001C. http://www.chargerlab.com/archives/536.html.
33. Qemu. https://www.qemu.org. Accessed: 2018-12-04.
34. QEMU with WIP TrustZone Support. https://git.linaro.org/virtualization/qemu-tz.git.
35. Shared memory size bigger than 1MB. https://github.com/OP-TEE/optee_os/issues/1523.
36. Stress-NG. https://kernel.ubuntu.com/~cking/stress-ng/. Accessed: 2019-20-01.
37. TEE_BigIntAdd fails when dest=op OP-TEE OS Issue #2577. https://github.com/OP-TEE/optee_os/issues/2577.
38. TRUSTSONIC. https://www.trustonic.com/solutions/trustonic-solutions-iot.
39. Using more than 1Mb with TEE_Malloc. https://github.com/OP-TEE/optee_os/issues/2178.
40. VMware ESXi. https://www.vmware.com/products/esxi-and-esx.html.
41. Workloads and governor effects. https://www.ibm.com/developerworks/library/l-cpufreq-3/.
42. ARM. ARM® CoreLink™ TZC-400 TrustZone®Address Space Controller. 2014.
43. ARM Limited. SMC CALLING CONVENTION System Software on ARM® Platforms. 2016.
44. M. Barbosa, S. B. Mokhtar, P. Felber, F. Maia, M. Matos, R. Oliveira, E. Riviere, V. Schiavoni, and S. Voulgaris. SAFETHINGS: Data Security by Design in the IoT. In *Dependable Computing Conference (EDCC), 2017 13th European*, pages 117–120. IEEE, 2017.
45. H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupé, and G.-J. Ahn. Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 441–452, New York, NY, USA, 2018. ACM.
46. Dominik Brodowski. CPU frequency and voltage scaling code in the Linux(tm) kernel.
47. Gartner. Leading the IoT Gartner Insights on How to Lead in a Connected World. 2017.
48. P. Greenhalgh. big.LITTLE processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17, 2011.
49. Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM trustzone. In *In Proc. of the 26th USENIX Security Symposium*, 2017.
50. M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. SeCloak: ARM Trustzone-based Mobile Peripheral Control. pages 1–13, 06 2018.
51. M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. *arXiv preprint arXiv:1805.04956*, 2018.
52. B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan. Open-TEE–An Open Virtual Trusted Execution Environment. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA-Volume 01*, pages 400–407. IEEE Computer Society, 2015.
53. ncc group. Implementing practical electrical glitching attacks, 2015.
54. nVidia. TRUSTED LITTLE KERNEL (TLK) FOR TEGRA: FOSS EDITION. 2015.
55. A. K. Reddy, P. Paramasivam, and P. B. Vemula. Mobile secure data protection using eMMC RPMB partition. In *Computing and Network Communications (CoCoNet), 2015 International Conference on*, pages 946–950. IEEE, 2015.
56. G. Technology. GlobalPlatform TEE Client API Specification v1.0.
57. G. Technology. TEE Internal Core API Specification Version 1.1.2.50. 2018.

A Appendix: Extending the Kernel

First, a new file containing the syscall used to retrieve the processor temperature `getcputemp` is created.

```
// populates temp with the CPU temperature in [m degC]
SYSCALL_DEFINE1(getcputemp, unsigned long *, temp)
{
  struct thermal_zone_device *tzd;
  // The name "bcm2835_thermal" is obtained
  // from /sys/class/thermal/thermal_zone0/type
  tzd = thermal_zone_get_zone_by_name("bcm2835_thermal");
  if (IS_ERR(tzd))
    return 1;
  thermal_zone_get_temp(tzd, &temp);
  return 0;
}
```

Listing 1.1: linux/custom/custom.c

This file must be referenced in the main kernel Makefile:

```
core-y += kernel/ [...] custom/
```

Listing 1.2: linux/custom/Makefile

The syscall must be included in `syscalls.h`:

```
asmlinkage long sys_getcputemp(unsigned long *temp);
```

Listing 1.3: linux/include/linux/syscalls.h

The `CALL` macro is used in `unistd.h`:

```
CALL(sys_getcputemp)
```

Listing 1.4: linux/arch/arm/kernel/calls.S

Use the next available syscall identifier:

```
#define __NR_getcputemp (__NR_SYSCALL_BASE+394)
```

Listing 1.5: linux/arch/arm/include/uapi/asm/unistd.h

In the following file and in addition to the modification listed above, note that *__NR_syscalls* must be incremented by one.

```
#define __NR_getcputemp 288
__SYSCALL(__NR_getcputemp, sys_getcputemp)
```

Listing 1.6: linux/include/uapi/asm-generic/unistd.h

At this point the new syscall is available to all user-mode applications running in TEE (Figure 2-❸ and Figure 2-❹). This syscall is then exposed in the REE kernel, tee-supplicant and the TEE kernel as if it were an official GlobalPlatform's API function definition.

```
unsigned long TEE_GetCpuTemperature(void);
```

Listing 1.7: optee_os/lib/libutee/include/tee_api.h

The *__NR_syscalls* value must be modified to account for the new syscall:

```
#define __NR_syscalls <INCREASE_BY_ONE>
```

Listing 1.8: linux/arch/arm/include/asm/unistd.h

The TEE function is a wrapper for the corresponding libutee implementation:

```
unsigned long TEE_GetCpuTemperature(void)
{
	unsigned long ret;

	TEE_Result res = utee_get_temperature(&ret);

	if (res != TEE_SUCCESS)
		TEE_Panic(res);

	return ret;
}
```

Listing 1.9: optee_os/lib/libutee/tee_api.c

TEE_SCN_MAX must also be increased accordingly and the call is given the next unique identifier (71 in our case):

```
#define TEE_SCN_GET_TEMPERATURE 71
#define TEE_SCN_MAX <INCREASE_BY_ONE>
```

Listing 1.10: optee_os/lib/libutee/include/tee_syscall_numbers.h

The utee syscall is declared in `utee_syscalls.h` and linked to its unique identifier:

```
TEE_Result utee_get_temperature(unsigned long *temp);
```

Listing 1.11: optee_os/lib/libutee/include/utee_syscalls.h

```
UTEE_SYSCALL utee_get_temperature, TEE_SCN_GET_TEMPERATURE, 1
```

Listing 1.12: optee_os/lib/libutee/arch/arm/utee_syscalls_asm.S

Add the syscall entry in `arch_svc.c`. The trailing comma is required.

```
SYSCALL_ENTRY(syscall_get_temperature),
```

Listing 1.13: optee_os/core/arch/arm/tee/arch_svc.c

```
TEE_Result syscall_get_temperature(unsigned long *temp);
```

Listing 1.14: optee_os/core/include/tee/tee_svc.h

This function serves as a wrapper to the REE kernel syscall used to retrieve the temperature:

```
#include <kernel/tee_temperature.h>
TEE_Result syscall_get_temperature(unsigned long *temp)
{
	tee_ta_get_temperature(temp);

	return TEE_SUCCESS;
}
```

Listing 1.15: optee_os/core/tee/tee_svc.c

A new file is created:

```
#ifndef TEE_TEMPERATURE_H
#define TEE_TEMPERATURE_H

#include "tee_api_types.h"

TEE_Result tee_ta_get_temperature(unsigned long *temp);

#endif
```

Listing 1.16: optee_os/core/include/kernel/tee_temperature.h

This function is called in ❽ and triggers a REE world switch ❼:

```
#include <compiler.h>
#include <string.h>
#include <stdlib.h>
#include <optee_msg.h>
#include <kernel/thread.h>
#include <kernel/tee_temperature.h>

TEE_Result tee_ta_get_temperature(unsigned long *temp)
{
	TEE_Result res;
	struct optee_msg_param params;

	memset(&params, 0, sizeof(params));
	params.attr = OPTEE_MSG_ATTR_TYPE_VALUE_OUTPUT;
	res = thread_rpc_cmd(OPTEE_MSG_RPC_CMD_GET_TEMPERATURE, 1, &params);

	if (res == TEE_SUCCESS) {
		*temp = params.u.value.a;
	}

	return res;
}
```

Listing 1.17: optee_os/core/arch/arm/kernel/tee_temperature.c

The following line in added in `sub.mk`:

```
srcs-y += tee_temperature.c
```

Listing 1.18: optee_os/core/arch/arm/kernel/sub.mk

A new message used to retrieve the temperature via RPC is declared:

```
// [out] temperature
#define OPTEE_MSG_RPC_CMD_GET_TEMPERATURE 21
```

Listing 1.19: optee_os/core/include/optee_msg.h

The same is done in another file:

```
// [out] temperature
#define OPTEE_MSG_RPC_CMD_GET_TEMPERATURE 21
```

Listing 1.20: linux/drivers/tee/optee/optee_msg.h

This function is declared inside the REE kernel:

```
#include <linux/syscalls.h>

static void handle_get_temperature(struct optee_msg_arg *arg)
{
	unsigned long cputemperature;

	// Linux kernel syscall
	if (sys_getcputemp(&cputemperature)) {
		arg->ret = TEEC_ERROR_GENERIC;
		return;
	}

	arg->params[0].u.value.a = cputemperature;
	arg->ret = TEEC_SUCCESS;
	return;
}
```

Listing 1.21: linux/drivers/tee/optee/rpc.c

In the same file, `handle_rpc_func_cmd` is modified by adding a case to handle the new RPC request:

```
case OPTEE_MSG_RPC_CMD_GET_TEMPERATURE:
		handle_get_temperature(arg);
break;
```

Listing 1.22: linux/drivers/tee/optee/rpc.c

After rebuilding the TEE client and kernel, the new syscall can be used as such from any TA ❾:

```
float tempC = TEE_GetCpuTemperature() / 1000.0f;
```

Listing 1.23: Usage from TA

This solution perfectly illustrates a workaround to the starvation of the REE world caused by the execution of the TEE.

In order to accomplish the secure storage micro benchmark, it was required to measure monotonic time, store and retrieve these measurements. The common denominator between the TEE kernel and the host application is the REE kernel. For this reason, it was decided to store measurements in the REE kernel, from which they could be gathered by the host application. Three syscalls were added in the REE kernel and made available in the TEE kernel.

These are first declared:

```
#define __NR_ktraceadd (__NR_SYSCALL_BASE+396)
#define __NR_ktraceget (__NR_SYSCALL_BASE+397)
#define __NR_ktracereset (__NR_SYSCALL_BASE+398)
```

Listing 1.24: linux/arch/arm/include/uapi/asm/unistd.h

```
CALL(sys_ktraceadd)
CALL(sys_ktraceget)
CALL(sys_ktracereset)
```

Listing 1.25: linux/arch/arm/kernel/calls.S

```
// save the current time as the specified id
asmlinkage long sys_ktraceadd(unsigned long id);
// returns the id+sec+ns of the requested index
asmlinkage long sys_ktraceget(unsigned long index, unsigned long* id,
                              unsigned long* sec, unsigned long* ns);
asmlinkage long sys_ktracereset(void);
```

Listing 1.26: linux/include/linux/syscalls.h

Implementation is stored in a separate file:

```
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/timekeeping.h>
#include <linux/slab.h>

#define MAX_KTRACE_ENTRIES 30
unsigned char ktrace_entries = 0;

struct ktraceadd_e {
       unsigned long id;
       struct timespec64 ts;
};

struct ktraceadd_e* ktraceadd_d;

SYSCALL_DEFINE1(ktraceadd, unsigned long, id)
{
       struct timespec64 ts;
       ts = ns_to_timespec64(ktime_get_ns());

       if (!ktraceadd_d) {
              ktraceadd_d = kmalloc(sizeof(struct ktraceadd_e)*MAX_KTRACE_ENTRIES,
                                    GFP_KERNEL | GFP_NOWAIT);
       }

       if (ktrace_entries < MAX_KTRACE_ENTRIES) {
              memcpy((void*)&ktraceadd_d[ktrace_entries].id, (void*)&id, sizeof(unsigned
                   ↪ long));
              memcpy((void*)&ktraceadd_d[ktrace_entries].ts, (void*)&ts, sizeof(struct
                   ↪ timespec64));
              ktrace_entries++;
              return 0;
       }

       return 1;
}

SYSCALL_DEFINE4(ktraceget, unsigned long, index, unsigned long*, id, unsigned long*,
      ↪ sec, unsigned long*, ns)
{
       if (ktraceadd_d && index >= 0 && index < ktrace_entries) {
              *id = ktraceadd_d[index].id;
              *sec = ktraceadd_d[index].ts.tv_sec;
              *ns = ktraceadd_d[index].ts.tv_nsec;
              return 0;
       }

       return 1;
}

SYSCALL_DEFINE0(ktracereset)
{
       ktrace_entries = 0;
       return 0;
}
```

Listing 1.27: linux/custom/custom.c

Next, three available syscalls identifiers are used. In the same file, *__NR_syscalls* must be incremented by three.

```
#define __NR_ktraceadd 290
__SYSCALL(__NR_ktraceadd, sys_ktraceadd)
#define __NR_ktraceget 291
__SYSCALL(__NR_ktraceget, sys_ktraceget)
#define __NR_ktracereset 292
__SYSCALL(__NR_ktracereset, sys_ktracereset)
```

Listing 1.28: linux/include/uapi/asm-generic/unistd.h

The *__NR_syscalls* value must be modified to account for the new syscalls:

```
#define __NR_syscalls <INCREASE_BY_THREE>
```

Listing 1.29: linux/arch/arm/include/asm/unistd.h

These functions can now be invoked from any REE user-mode application. Instrumentation tests *test1* and *test2* are added directly from the host application using ❸, and then retrieved and displayed.

```
#include <unistd.h>
#include <sys/syscall.h>
#include <time.h>

// As defined in
// optee/linux/include/uapi/asm-generic/unistd.h
#define SYSCALL_KTRCEADD 290
#define SYSCALL_KTRCEGET 291
#define SYSCALL_KTRCERESET 292

printf("Calling␣SYSCALL_KTRCERESET\n");
syscall(SYSCALL_KTRCERESET);

printf("Calling␣SYSCALL_KTRCEADD␣0\n");
syscall(SYSCALL_KTRCEADD, "test1");
sleep(1);
printf("Calling␣SYSCALL_KTRCEADD␣1\n");
syscall(SYSCALL_KTRCEADD, "test2");

char kget_name[20];
unsigned long kget_sec;
unsigned long kget_ns;

for (int i = 0; i < 2; ++i) {
       syscall(SYSCALL_KTRCEGET, i, &kget_name, &kget_sec, &kget_ns);
       printf("SYSCALL_KTRCEGET␣index␣%d:␣name=%s␣sec=%ld␣ns=%ld\n",
              i, kget_name, kget_sec, kget_ns);
}
```

Listing 1.30: Host application usage example

Trace calls can then be added anywhere in the REE core ❽. Once called, an RPC is made ❼ to the REE kernel. For example:

```
#include <kernel/tee_ktrace.h>

static TEE_Result ree_fs_create([...])
```

```
4 {
5       TEE_Result res;

7       // Measuring instrumentation delay
8       tee_ta_add_ktrace(200);
9       tee_ta_add_ktrace(201);

11      // Measuring time taken to enter the monitor
12      mutex_lock(&ree_fs_mutex);
13      tee_ta_add_ktrace(202);

15      // [...]
16 }
```

Listing 1.31: optee_os/core/tee/tee_ree_fs.c