



Article

MINITEE—A Lightweight TrustZone-Assisted TEE for Real-Time Systems

Songran Liu ^{1,2} , Nan Guan ², Zhishan Guo ^{3,*}  and Wang Yi ¹

¹ College of Computer Science and Engineering, Northeastern University, Shenyang 110819, China; liusongran@stumail.neu.edu.cn (S.L.); wangyi@ise.neu.edu.cn (W.Y.)

² Department of Computing, The Hong Kong Polytechnic University, Hong Kong 999077, China; nan.guan@polyu.edu.hk

³ Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816-2362, USA

* Correspondence: zsguo@ucf.edu

Received: 31 May 2020; Accepted: 8 July 2020; Published: 11 July 2020

Abstract: While trusted execution environments (TEEs) provide industry standard security and isolation, TEE requests through secure monitor calls (SMCs) attribute to large time overhead and weakened temporal predictability. Moreover, as current available TEE solutions are designed for Linux and/or Android initially, it will encounter many constraints (e.g., driver libraries incompatible, large memory footprint, etc.) when integrating with low-end Real-Time Operating Systems, RTOSs. In this paper, we present MINITEE to understand, evaluate and discuss the benefits and limitations when integrating TrustZone-assisted TEEs with RTOSs. We demonstrate how MINITEE can be adequately exploited for meeting the real-time needs, while presenting a low performance overhead to the rich OSs (i.e., low-end RTOSs).

Keywords: real-time system; ARM TrustZone; trusted execution environment

1. Introduction

As real-time embedded systems become more complex and interconnected, certain sections or parts of the systems may need to be protected to prevent unauthorized access, or isolated to ensure functional/non-functional correctness. For example, consider an autonomous delivery drone used to transport packages to remote locations. The drone may communicate with the base station via real-time communication protocols to update mission-related information (e.g., new customer location). If those information are tampered with an ill-intentioned entity, the entire drone can be mis-routed, leading failed delivery [1]. Complex authentication mechanisms can improve system security, but at the price of introducing large overhead and degrading the real-time performance of the system [2]. Therefore, a platform-level security solution for such systems should not only enhance the security but also minimize the negative influence to the real-time performance.

Trusted Execution Environments (TEEs) [3] are widely used to power the security of embedded systems. TEE shields sensitive information in an execution environment which runs in isolation from the main operating system. Nowadays, ARM TrustZone-assisted [4] TEEs (as we focus on TrustZone-assisted TEEs in this work, we use TrustZone-assisted TEEs and TEEs interchangeably in the remaining of the paper) have been extensively deployed for mobile devices to protect security-critical data like cryptographic key and payment information [5]. The wide spread availability of TEE in commercial-off-the-shelf (COTS) systems (1) dramatically reduces the need for the development and manufacturing of hardware solutions such as encrypted hardware [6], code obfuscation [7] and so forth, and (2) allows quick re-deployment as TEEs are isolated with the main operating system.

However, applying current TEE solutions to real-time systems is not straightforward. Firstly, TEEs, though supported by hardware, still create significant time overhead. Specifically, each instance of TEE execution is initiated by a setup phase and exits through a destroy phase. Since TEE leverages architecture-specific Secure Monitor Calls (SMCs) to realize those phases, setting up and tearing down a TEE session requires 7700 μ s based on our measurements on a Zedboard platform. Furthermore, as TEE design specifications are completely devoted to security requirements, existing TEEs do not contemplate real-time properties during design time. For instance, data/instruction fetches and write-backs during TEE execution may cause large variation on task execution time, thereby weakening time predictability. Finally, most TEE solutions are initially designed for Linux or Android, which brings many implementation constraints (e.g., driver libraries incompatibility, etc.) when integrating with low-end RTOS (e.g., FreeRTOS: <https://www.freertos.org>, NuttX: <https://nuttx.org/>, etc.). More details are shown in our motivated example in Section 3.

To overcome the aforementioned shortcomings, we design and implement a new TEE alternative for RTOS, named MINITEE. To alleviate the time overhead and unpredictability, we optimize the TEE service management rules. To minimize the memory footprint, we implemented a lightweight kernel to manage Trusted Applications (TAs) residing in Secure world and provide services such as interrupt handling and memory management. Specifically, rather than setting up and destroying a TEE session in each calling iteration, we resume and suspend the TEE session at the begin and end of each invocation. In such a way, MINITEE can simplify the calling phases and avoid frequent memory allocation and reclamation. To guarantee the compatibility with low-end RTOS, we also design a TEE-related driver library. To understand and evaluate how RTOS (especially low-end RTOS) can incorporate with TEE to enhance the system security level, we conducted extensive experiments which demonstrate the overhead introduced by MINITEE to original RTOS.

The technical contributions of this paper can be summarized as follows:

- We comprehensively discussed the obstacle of combining RTOS with current TEE solutions and propose a RTOS-friendly TEE solution with predictable and bounded overhead;
- We present a case study on a 3DOF control system to demonstrate the use case of MINITEE;
- We conduct comprehensive evaluation on a realistic hardware platform, focusing on the overhead incurred by MINITEE on the real-time properties to a legacy RTOS.

The rest of this paper is organized as follows. In Section 2, there is a brief introduction of the ARM TrustZone technology and TEEs. After that, in Section 3, a motivation discussion is given, followed by the design principles. Sections 4 and 5 details the implementation of MINITEE. Case study and evaluations are presented in Section 6. In Section 7, a set of related work are given and discussed. Finally, Section 8 summarizes this work's findings and directions for future works.

2. Background

2.1. Overview of TrustZone

ARM TrustZone technology [4] refers to a hardware security mechanism introduced since ARMv6 architecture, which includes security extensions to ARM System-On-Chip (SoC) covering the processor, memory and peripherals. Recently, ARM also extends TrustZone to ARMv8-M architecture for the Cortex-M processor family [8]. TrustZone for ARMv8-M has the same high-level features as TrustZone for applications processors, with the benefit that context switching between both worlds is done in hardware for faster transitions. In the remainder of this work, when describing TrustZone, we focus on the specificities of this technology for Cortex-A processors. The distinctive aspects of TrustZone for ARMv8-M are out of the scope of this paper.

From the hardware level, TrustZone splits the processors into two execution environments, the Normal World and the Secure World (as shown in Figure 1). A new 33rd processor bit (for 32-bit processors), the Non-Secure (NS) bit, indicates in which world the processor is currently executing, and is propagated over the memory and peripheral buses. To ensure a strong isolation between

Secure and Normal Worlds, some special registers are banked, such as a number of System Control Co-processor (CP15) registers. Typically, those two worlds can transit to each other through an extra processor mode: the monitor mode. And during world switching, there is an additional cost associated with having to save and restore un-banked processor registers of each world. Moreover, both worlds have their own user space and kernel space, together with cache, memory and other resources.

Software stacks in the two worlds can be bridged via a new privileged instruction – Secure Monitor Call (SMC) running in monitor mode. The Normal World software cannot access the Secure World’s resources while the latter can access all the resources. Base on this asymmetrical permission, the Normal World provides a Rich Execution Environment (REE) based on a feature-rich OS. Meanwhile, the Secure World, always locates a small secure kernel, which provides a Trusted Execution Environment (TEE).

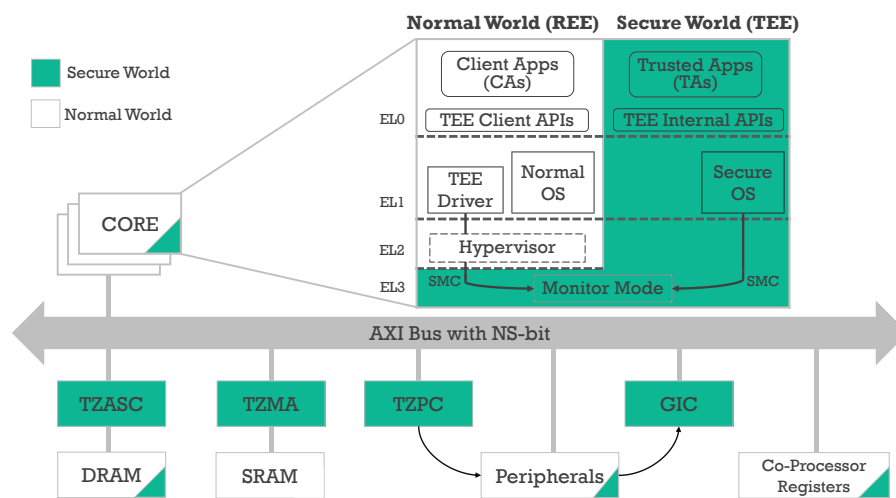


Figure 1. A general architecture on a TrustZone-assisted System-On-Chip (SoC): TrustZone splits CPU into Normal World and Secure World, all other hardware resources are split as well. Each world has its own user and kernel space, and can switch to each other by SMC instruction. TrustZone Address Space Controller (TZASC): configure DRAM as secure or non-secure (S/N) partition. TrustZone Memory Adapter (TZMA): configure SRAM S/N partition. TrustZone Protection Controller (TZPC): configure peripheral S/N partition. General Interrupt Controller (GIC): control interrupt, can also configure interrupt S/N partition.

2.2. Structure of a TEE

As described in the previous parts, TrustZone has been widely used as a cornerstone hardware technology for enabling TEE on ARM-based platforms. However, the TEE standardization (e.g., TEE client and internal API standards) is still in its infancy. For example, an application written for one TrustZone-based TEE will generally not be able to run on another. As they may be using a different TEE OSs or REE OS drivers. More details can be found in survey [9]. One initiative in TEE standardization has been undertaken by GlobalPlatform (GP) [10], which provides a widely adopted standard in TEE development community. The TEE Client API [11] is a very generic and thin layer consisting of a small number of functions and definitions that allow the transfer of data back and forth from the REE to a TA.

2.3. TEE Augmented Real-Time Task Model

We assume a set of independent sporadic real-time tasks Γ running on a single TrustZone-enabled processor, which can switch between Secure and Normal World through a Secure Monitor Call (SMC) instruction. In addition, the executing code of each real-time task τ_i is given in the form of code segments, where $Seg_i^{N_1}$ and $Seg_i^{N_2}$ are the code segments executing in Normal World, and $Seg_i^{S_1}$ is the code segment executing in Secure World. As shown in Figure 2, for a task τ_i with TEE requirements

(i.e., executing a few code segments in Secure World), segments $Seg_i^{N_1}$ and $Seg_i^{N_2}$ are interleaved by segment $Seg_i^{S_1}$. TEE-specific APIs are invoked before and after each secure execution segment to bridge the communication with normal execution segments. Similarly, if a task does not require services in TEE, it only has a single normal execution segment $Seg_i^{N_1}$.

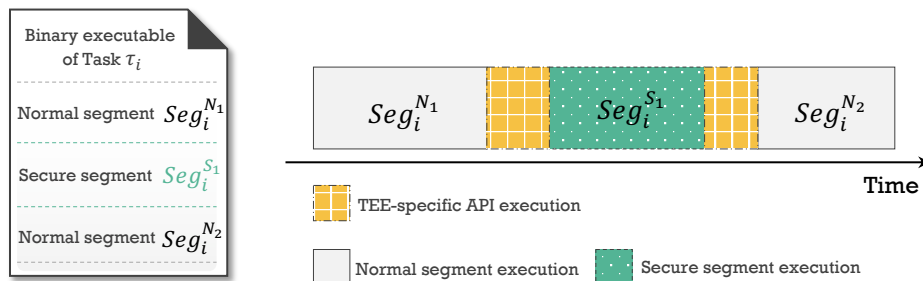


Figure 2. Trusted Execution Environment (TEE) augmented real-time task model.

Note that, we assume the code segments (both Seg_i^N and Seg_i^S) of each task are given in advance. This can be achieved by a TEE-aware application development or utilizing a TEE-aware program slicing tool [12] to refactor the legacy real-time applications. To keep consistency with normal real-time system terminology, we still called Seg_i^N executing in Normal World as real-time task τ_i , and segments Seg_i^S in Secure World as Trusted applications (TAs). Also we assume one real-time task can invoke multiple TAs in a sequential manner, and multiple real-time tasks could invoke the same TAs.

As an example, consider a potential use case for TEEs: the flight control system. The real-time tasks would be responsible for the majority of the functionalities: sensor reading, network management, mission planning, and so forth. While a TA (i.e., secure segment) would be used to—decrypt set-point information received from base station, or encrypt images captured by a local camera.

3. Motivation Discussion and Design Principles

In this section, we first discuss the motivation behind MINITEE by evaluating the TEE-introduced performance overhead, and then describe the design principles behind MINITEE.

3.1. Motivation

Efficiency Concerns. As current TEE solutions being completely devoted to security requirements, and TEE specifications do not contemplate real-time properties on the Secure World OS side [13–15]. To further understand the time overhead introduced by TEEs, we run the Linux RT_PREEMPT kernel v3.14 (as Normal World OS) alongside OP-TEE: OP-TEE currently is the most developed open-source TEE solution in compliance with GP specifications. Also, OP-TEE is widely adopted by research groups. OS v2.3.0 (as Secure OS) on the Xilinx Zedboard. Since TEE leverages architecture-specific secure monitor calls (SMCs) to realize the world communication, the time overhead associated with TEE execution, including setting up and tearing down a TEE session, requires 7,700 μ s. Similar results are also reported in References [16,17]. Apparently, those large time overhead and highly variable execution time can heavily impair the predictability of RTOS.

To further understand the source of overhead in current TEE design (e.g., OP-TEE), we try to break the TA-invoking (i.e., calling TA service in OP-TEE OS) procedure into contiguous sub-processes which is shown in Figure 3. As the illustration, when a task τ_i triggers a TA-invoking request, it will sleep until the invocation finishes its execution and answers back. In general, the TA-invoking procedure is constituted by five essential parts (the sequence may be different), which are:

- (1) Normal World task issues a TA-invoking request by calling the TEE client APIs. Then TEE driver will trap the CPU into monitor mode to communicate with OP-TEE kernel to initialize a private communication channel (i.e., TEE context) for passing messages.
- (2) OP-TEE kernel will create a TA execution environment (e.g., session control block, stack space) before invoking that TA function. Note that, if the called TA is busy (i.e., called by other Normal World tasks), OP-TEE kernel will suspend the newly arrived request. Since each TA in OP-TEE has an unique private stack.
- (3) The invoked TA starts to execute.
- (4) OP-TEE issues a SMC to get the CPU back to Normal World after destroying the TA execution environment and reclaiming the memory space.
- (5) After replying the TA's execution results to the calling task, TEE driver tears down the TEE context.

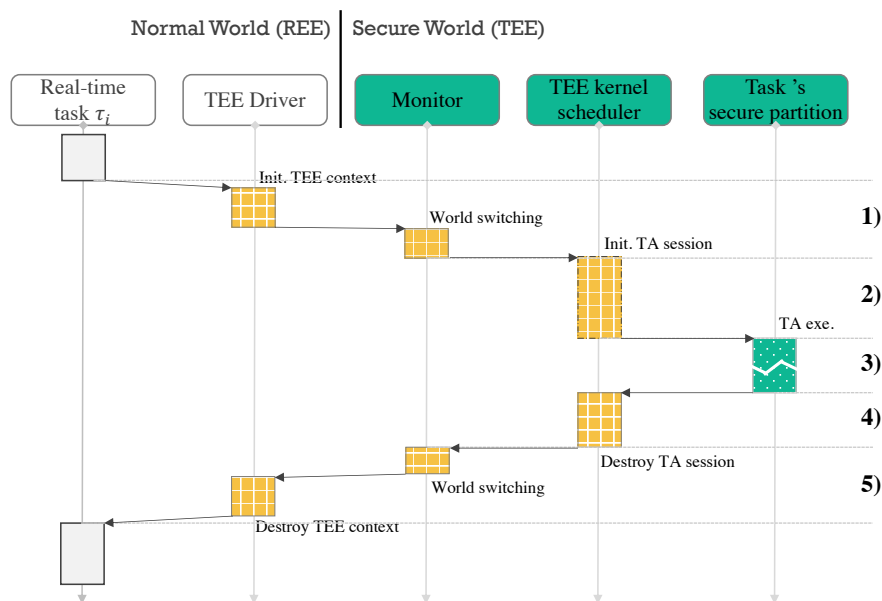


Figure 3. TEE workflow breakdown.

From the above description, we can draw the following observations: frequent memory allocation/reclamation will introduce more run-time overhead and fragment memory pool quickly, thereby resulting in large execution time variation. Moreover, higher-priority tasks may be blocked by lower-priority ones during TA invocation resulting in a priority-inversion [18], making the whole system unschedulable.

Security Concerns. Each communication must not lead to or propagate hazards and faults. When communicating with TEE, the TA-invoking request may lead to suspension of the related task, or OS switching, so sufficient mechanisms must be prepared to cope with various exceptions such as time out and state restoring. Considering multi tasks sharing the same TEE, side-channel information leakage should be carefully removed. Some examples [19,20] show how to leverage the communication and vulnerabilities to attack the TrustZone protection. Besides, too many communication requests may trigger DoS (Denial of Service) of related RTOSs.

Implementation Concerns. We also measured the memory footprint of some open-source TEE solutions (i.e., OP-TEE and SierraTEE: <https://www.sierraware.com>) by comparing with a low-end RTOS (i.e., FreeRTOS). Although OP-TEE take the consideration of minimizing the Trusted Computing Base (TCB) size at the design time, it is still about 6 times larger than FreeRTOS, while SierraTEE is about 4 times larger than FreeRTOS kernel.

3.2. Design Principles

Motivated by the above limitations, we can draw the following design principles that should be satisfied by MINITEE.

- **P1. Predictable and Bounded Run-time Overhead:** Performance overheads on both the Normal and Secure Worlds incurred by MINITEE should be Predictable and bounded. Otherwise, it will hurt the real-time properties.
- **P2. Security Guarantees:** Each communication must not lead to or propagate hazards and faults, And the shared MINITEE resources should not open any side-channel for malicious tasks.
- **P3. Minimize Trusted Computing Base:** On the one hand, large software size may potentially open more attack surfaces to adversaries. Relying on the hardware support of TrustZone technology as much as possible, will definitely help us minimize the trusted computing base of the system and, consequently, decreasing the attack surfaces to adversaries. On the other hand large code size may obstacle the usage in resource-constrained devices.

4. Design

In this section, we first describe the overview of MINITEE, including its architecture and high-level workflow. Next, we detail the design of each MINITEE component.

4.1. Overview

Figure 4 illustrates the overall architecture. At the core lies the two basic software components—MINITEE-driver and MINITEE. MINITEE-driver, placed within a Normal World RTOS, provides an interface for real-time tasks to leverage MINITEE. The main components of MINITEE are the Monitor and MINITEE kernel. The Monitor is responsible for the switching between the Secure World and the Normal World, including saving and restoring the context of the corresponding world, passing control-related command and data. The MINITEE kernel is responsible for providing the fundamental core services, such as *TEE task* management, memory management and interrupt handling.

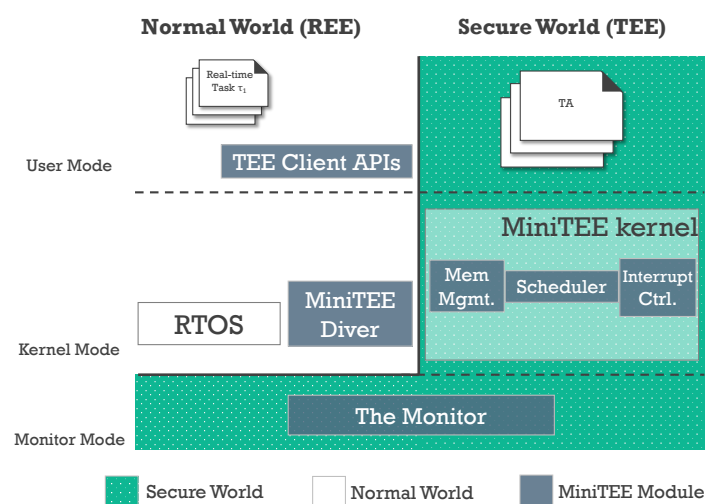


Figure 4. MiniTEE architecture.

The workflow of the MINITEE-enabled system is as follows—(i) When the system is booting up in monitor mode, the processor hands over to Secure World to initialize the TEE kernel, then switch back to Normal World to initialize the normal RTOS; (ii) When a real-time task in the Normal World RTOS requests a new security service (i.e., invoking its cooperative TA), MINITEE-driver forwards the request to the Monitor through SMC calls. Then the scheduler decides to created a *TEE task* according

to the command parameters or restore an *suspend TEE task* interrupted by Normal World events; and (iii) all the message data among the real-time tasks (in the Normal world) and the TAs (in the Secure world) are properly routed by shared memory.

4.2. Architecture Design

4.2.1. The Monitor

As the execution of TrustZone-based ARM CPU is split into two parts (i.e., Secure/Normal World) and some registers are not banked, MINITEE has to preserve the world context during the switching operation for the correctness of the whole system. This part is the most performance-related function of the system, as the worst-case response time (WCRT) of a real-time task and RTOS interrupt latency are directly affected by the time consumed for the context save/restore operations (called world switching in the following part of this work). For this reason, most of code executed during the world switching is written in ARM assembly in order to achieve the best performance, and all the maskable interrupts are disabled to guarantee the atomicity of the context switching operation.

In MINITEE, a world switching can be triggered in the following two main cases: one world requests a world switching (e.g., TA invoking, TA execution finished, etc.) by calling Monitor service through an SMC instruction; an interrupt assigned to the Normal/Secure World is triggered during the execution of the Secure/Normal World. In this context, The Monitor first saves the current state of the world suspended, then restores the state of the other world.

4.2.2. MINITEE Kernel

A. MINITEE Execution Model and TEE Tasks

In this part, we first discuss the design philosophy behind MINITEE kernel, then explain how MINITEE can achieve the principles presented in Section 3.

In MINITEE, TAs as basic components provide security-related services residing in the Secure World. As illustrated before (Figure 3), in most current security-oriented TEE design, TA-related execution environment will be created and destroyed at the beginning and end of a TA-invoking, which acts as the main contributor of run-time overhead and execution time uncertainty. Such a design can increase the physical memory utilization which fits well with general purpose systems (e.g., Linux) where average-case performance is the main target. However, the core design requirement of RTOS is always predictability. Frequent memory allocation and reclamation could potentially fragment the memory pool quickly, thereby introducing large variation in task execution time.

The key insight of our approach is to utilize the periodicity of real-time task model. That means the same TA may be invoked in a periodic manner by the same real-time task in Normal World RTOS. One potential solution is to trade off system predictability with memory space consumption, by keeping the TA execution environment alive after its execution rather than destroying it. The main concern of such design comes from the large memory consumption to allocate stack space for each TA. To address this issue, MINITEE introduces the concept of *TEE task*.

TEE Task: A *TEE task* is a lightweight and process-like structure (shown in Figure 5), which is used to manage resources (e.g., shared memory, stack) for its TA and track TEE invocation execution contexts when TEE needs to switch back to the REE before the invoked TA is completed.

To manage the run-time information, MINITEE kernel keeps track of each *TEE task* by maintaining a *TEE task control block (TTCB)*. TTCB holds the *state* and *owner* of the *TEE task*, a data buffer to store passing message data from Normal World in shared memory, pointers to the entry of current executing TA, to the bottom of the stack space using to store local variables of its calling TA(s). The *status* may fall into one of the following four states: *free*, *idle*, *running*, and *suspend* and the state transformation flow is described in Figure 6. Since a *TEE task* just acts as the proxy in MINITEE of a Normal World real-time task. Once MINITEE decides to response a request from real-time task τ_i , a *TEE task_i* will be

created and the corresponding attribute *owner* is set to τ_i . Thereafter, *TEE task_i* is used to handle all the requests from task τ_i .

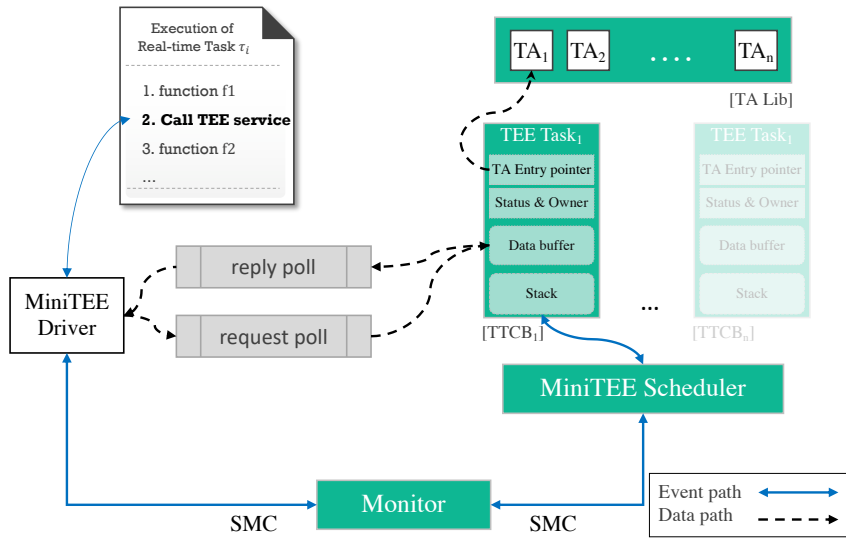


Figure 5. TEE task model.

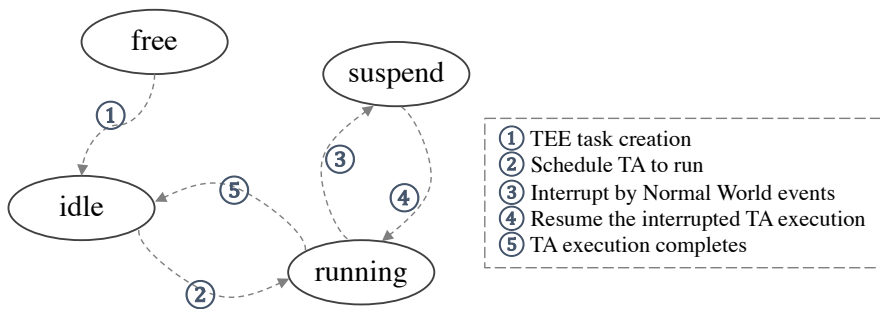


Figure 6. TEE task state transformation flow.

TEE Task Scheduling: The MINITEE kernel adopts the request-response execution model, where the service request is made from the Normal World (i.e., MINITEE driver called by a real-time task) and the response is performed by the Secure World (i.e., TEE Task). And MINITEE kernel does not implement any scheduling mechanism. This is mainly due to the role of TAs: TAs are not designed to be a standalone program, but library functions providing security-related services for Normal World real-time tasks. That is to say, the execution of the TEE task is tied to the execution of the caller real-time task which is under the RTOS scheduling decision. This means TEE tasks are scheduled by the RTOS kernel.

TEE Task Life-circle: As discussed before, to alleviate unpredictability and decrease time cost, MINITEE never reclaims the stack space of each TEE task since this TEE task is created. After TEE task finishing this round’s TA execution, MINITEE scheduler simply clears the whole stack so that it can be reused by the following called TAs, and changes the TTECB’s status to *idle*.

Next, we use an example to illustrate the TEE task run-time behavior, which is shown in Figure. At the very beginning, a real-time task τ_1 send a request to MINITEE to invoke the service provided by TA_1 . The MINITEE scheduler handles the request by first checking whether there already exists a TEE task being tied to task τ_1 . If not, a new TEE task (e.g., TEE task₁) will be created and the TTECB will be initialized with the passing message data in shared memory and status is changed from *free* to *idle*. In the next step, MINITEE scheduler modifies the *status* of this TTECB to *running* and calls the TA pointed by the pointer in field *TA entry pointer* of this TTECB. Once a higher priority task arrives

in Normal world (a related explanation is presented in Section 4.2.2-B), MINITEE scheduler saves the execution context onto *TEE task*₁'s stack and changes *status* to *suspend*, then handovers CPU to Monitor to perform a world switch. Once the CPU enters into Secure world again, MINITEE scheduler first check who issued this request. If it is task τ_1 , then previous execution of TA_1 will be resumed by restoring execution context from *TEE task*₁'s stack. After TA_1 completes his execution, MINITEE scheduler first sets *TEE task*₁'s *status* to *idle*, than wraps the results into a pre-defined data structure which will be located into the shared memory thereafter. Finally, MINITEE scheduler cleans up the stack space and return a *success* to Normal world.

B. MINITEE Memory Model

Both interrupts (e.g., IRQ) and exceptions (e.g., SMC call) can only carry data in the general registers, that is not enough for big amount of information transforming, so we need an efficient data path between Normal World real-time tasks and *TEE tasks*. MINITEE performs such communications via a Remote Procedure Call (RPC) style protocol.

- Shared-memory-based Data Channel—MINITEE uses the shared memory as the data channel, realized as two cache pools—a request pool and a reply pool.
- Channel Maintenance—The first 32 bytes (e.g., the first slot) of each poll is defined as the pool-head, which is used for maintenance of the pool. The head of request pool is partitioned into two fields—*request-task* corresponds to the message producer (aka, real-time tasks) and *msg-bitmap* identifies valid slots storing the message data. Note that, at one time, there is at most one real-time task's message occupying the request pool. Since before a *TEE task* running, the message will be copied to the *TEE task*'s private message space and destroy the duplicate message in request pool. The interrupts are disabled during such memory copy to avoid the occurrence another task's message (due to higher priority task preemption) in the request pool note that, the blocking time of such an atomic execution can be alleviated by using Direct Memory Access (DMA). In this way, the management pool is free from fragmentation and garbage collection, thereby enhancing the predictability of the whole system. Note that, the structure and working principle of the reply pool is similar to the request pool. The difference is that memory copy direction is from reply pool to calling task's local stack.
- Slot Size and Efficiency—Larger slot size does help for the increment of bandwidth, but it increases the latency of Larger slot size does help for the increment of the bandwidth, but it increases the latency of RPC yet. Note that the operation to read/write memory need considerable CPU cycles. So the slot size is the result of trade-off between bandwidth and latency. In MINITEE, each pool possesses a 4k bytes page (can be customized to a larger page if needed in future), and contains 128 message slots. The 32-byte message length coincides with the cache line size of the ARMv7 CPU, which makes a high efficiency of the multi-level caches in the CPU. With two distinguishable pools in an RPC channel, it is helpful to decrease the occurrence of page faults, hence increases the speed of RPC message accessing. Similar approach is also adopted in Reference [21].

C. MINITEE Interrupt Handling

As discussed in Section 1, our purpose is utilizing TEE architecture to secure or protect parts of traditional real-time applications. So in our design, the scheduler still resides in the Normal World. To keep reactive to the Normal World events (e.g., interrupts) and the preemption of higher priority task, MINITEE should be possible to guarantee the responsiveness to the Normal World events during the Secure World executing.

During Secure World execution, if a native interrupt (i.e., interrupt partitioned to Secure World) is received, MINITEE kernel will directly process it. On the opposite, if a foreign interrupt (i.e., interrupt belongs to Normal World) occurs, MINITEE will first save the context of current executing *TEE task* to its stack, and mask all interrupts, then issue an SMC call with a value of current interrupt ID. After the processor backs to Normal World and completes the interrupt handling, a new SMC command will be

issued to notify that MINITEE kernel could continue the latest interrupted *TEE task*. The handling mechanism of foreign interrupt for Normal World is similar.

D. MINITEE Security Design

In the terminology of information system security, risks exist if there are latent vulnerabilities and threats. For the perspective of security, the communication path and the target system are viewed as an entirety. If any part exists an exploitable vulnerability, the communication will lead to threats. Unfortunately, the design of communication can not give absolute immunity from attacks. For example, the key step of the attack in References [22] and [19] is to transfer some compliant data to trigger the buffer overflow in the Secure World kernel, and it is almost impossible for the communication mechanisms to detect such behaviors. For the sake of feasibility, we only consider lightweight measures and limited threats. Based on these ideas, our security design mainly counters two types of threats.

The first is the DoS attack threat. Any real-time tasks should not damage the availability of resources (e.g., CPU, TEE) in the whole system. More specifically, MINITEE as a shared resources provides security-related services for all legal real-time tasks. Although, current MINITEE design takes isolation and task preemption in mind. It is still possible for malicious tasks to launch DoS attack to influence the timing of higher priority task. Since in MINITEE some procedures (e.g., execution in monitor mode and memory copy) work under interrupt disabled. The deliberate attacks could amplify such blocking time by frequently calling MINITEE services, thereby affecting the normal execution of higher priority tasks. To this end, MINITEE sets up a counter for each created *TEE task* to test the calling frequency. If the calling frequency exceeds a pre-defined threshold, MINITEE will halt the response of that *TEE task*, and send warning message to Normal World RTOS.

Another thing is about authentication. In general, a real-time task only has the privilege to invoke service from a few TAs. To this end, we adopt similar method with OP-TEE to provide a pair of UUID for each real-time task and its calling TAs at the implementation stage. During run-time, only legal invocation (i.e., the UUID paired successfully) will be delivered and processed by MINITEE. One more thing should be noted is that, for the sake of responsiveness to Normal World activities, Normal World interrupts could disturb MINITEE execution. To reduce attack surface, only registered Normal World interrupts are available to signal CPU during MINITEE execution by configure the TrustZone-enabled GIC (General Interrupt Controller).

5. Implementation

We have implemented a MINITEE prototype on the Xilinx Zedboard platform with two Cortex-A9 processors (with one core disabled). This section describes the implementation of all the components behind MINITEE.

Normal World Implementation: One key challenge when developing TAs for TEEs from different vendors is the different set of available Application Programming Interfaces (APIs) and standard libraries. In an effort to increase interoperability between TEEs from different vendors, GlobalPlatform has specified a standard API [11] for their users running in Normal World. MINITEE implemented a subset of these GlobalPlatform APIs. Ideally this should enable source code compatibility between TAs written for MINITEE and other TEEs.

MINITEE Implementation: In the current implementation of monitor, the context of Normal World side is composed by 27 registers—13 General Purpose Registers (R0-R12), the Stack Pointer (SP), the Linker Register (LR) and Saved Program Status Register (SPSR) for each of the following modes—Supervisor, System, Abort and Undefined. The “high” General Purpose Registers (R8-R12), as well as the SP, LR and SPSR of the FIQ and IRQ modes are not included, as they are mutually exclusive for each world. Among the coprocessor registers, almost all of them are banked—only the SCTL and the ACTLR need to be preserved. As for the context of Secure World side, except 27 registers mentioned before, it also includes the registers used by Monitor mode.

6. Evaluation

Our solution was evaluated on a ZedBoard targeting a dual ARM Cortex-A9 running at 667 MHz. Firstly, we measured the binary size of MINITEE. Then, we presented a realistic case study to explain how MINITEE can be integrated with a low-end RTOS (e.g., FreeRTOS) to enhance the security of real-time applications. Lastly, in Section 6.3, we carefully measured the time overhead introduced by MINITEE.

6.1. Binary Size

In order to assess the memory footprint of each software component of the implemented architecture we used the size tool of the ARM GNU Tool-chain. We evaluated MINITEE, as well as the native version of FreeRTOS (v. 10.0.2). Table 1 presents the collected measurements, where boot code, libraries and drivers were not taken into consideration. The main reasons behind such a low memory footprint (compared with OP-TEE) are related to the principle of minimal implementation followed during MiniTEE design which relies on the careful design and static configuration of each MINITEE component.

Table 1. MINITEE memory footprint (in bytes).

Software	Memory Footprint			
	.text	.data	.bss	Total
MINITEE	5854	10	235	6099
FreeRTOS(v10.0.2)	20,543	20	1036	21,599
OP-TEE	74,158	328	43,920	11,8406

6.2. Case Study

(1) 3DOF Control System Overview

In order to motivate and evaluate the presented research, we use the example of a flight control system for a 3-Degree of Freedom (3DOF) helicopter. This demonstrative system runs many of the same types of tasks which could be expected to run on an Unmanned Aerial Vehicle (UAV) surveillance system (shown in Figure 7). The Electronic Control Unit (ECU) communicates locally with sensors and actuators, as well as a camera module on the 3DOF helicopter. Moreover, the ECU uses TCP/IP protocol to exchange information with a base station. We assume the whole ECU system is combine by three parties, camera task, network task and control task. One or more sub-functions in each task have some degree of protected data (e.g., encryption keys).

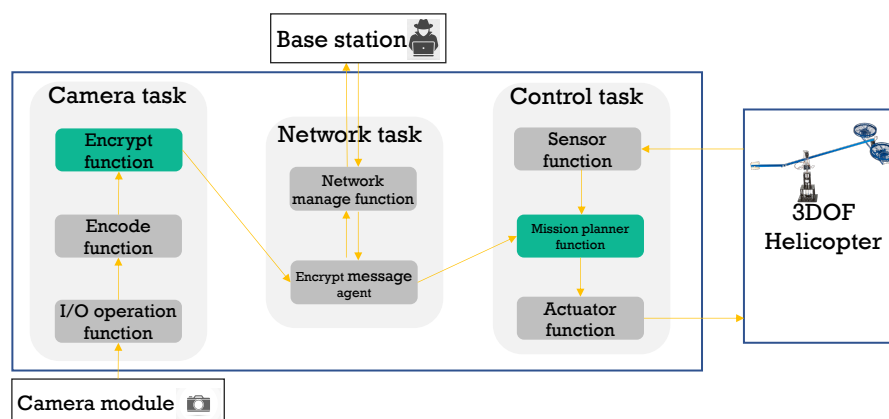


Figure 7. 3-Degree of Freedom (3DOF) control system outline.

Camera task is used for image processing. The I/O operation function mimics the behavior of a camera driver; to perform repeatable experiments, our system simply places a fixed set of images into a Memory File System (MFS) and extracts them in order. The encoder function is realized as a JPEG compressor. Encryption function uses the AES ECB cipher using a protected, secret 128bit key. The encrypted image is then passed to the encrypt message function in *Network task*. *Network task* is responsible for communicating with base station. The encrypt message function, on the one hand, receives encrypt data from *Camera task* then pass them to the base station via network manager function, on the other hand, receives encrypted set-point value from base station and send to Control task. *Control task* is responsible for the control subsystem. The sensor function receives and parses incoming sensor data for the other tasks. The mission planner function computes the control output based on measured sensor data, then move the 3DOF towards the set-point parsed by decrypt function. Finally, the actuator function prepares the actual output commands and send them to physical actuators.

(2) 3DOF Control System Implementation

Our control system runs on one of the processor cores on ZedBoard (shown in Figure 8a). The 3-Degree of Freedom (3DOF) helicopter (displayed in Figure 8b) is a simplified helicopter model, ideally suited to test intermediate to advanced control concepts and theories relevant to real-world applications of flight dynamics and control. It is equipped with two motors that can generate force in the upward and downward direction, according to the given actuation voltage. It also has three sensors to measure elevation, pitch, and travel angle as shown in Figure 8b). All three tasks are implemented on FreeRTOS (v 10.0.2) under 0.1 ms sysTick rate. And the communication among those three tasks can use non-blocking queue API provided by FreeRTOS kernel to avoid long blocking times. As the baseline, all the sub-functions in the whole task set run in the normal world without ARM TrustZone enabled. For the MiniTEE version, the sub-functions wrapped with green blocks (as shown in Figure 7) are treated as TAs running in secure world.

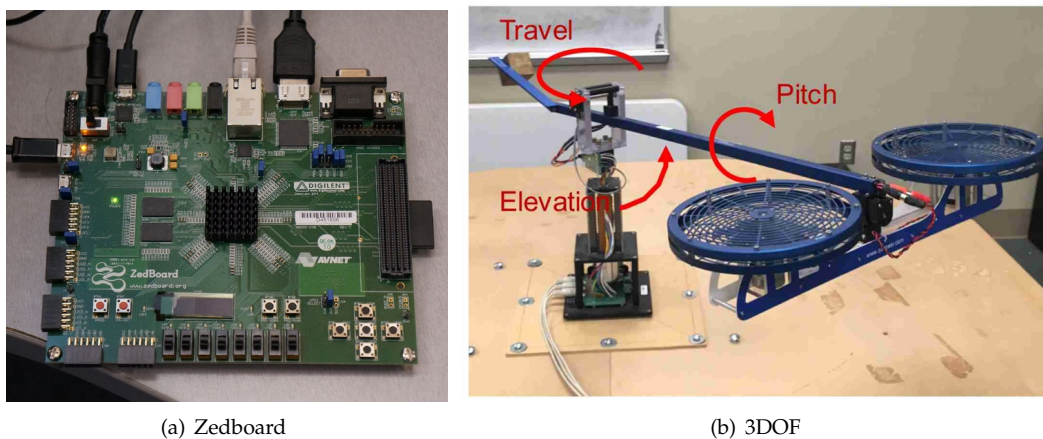


Figure 8. 3DOF control system implementation with Zedboard.

(3) Results

Demo system booting time: As discussed in Section 4, MINITEE creates all the *TEE tasks* after each platform reboot. This design may potentially harm system booting up time. In this particular demo system, we measured the overhead by comparing following two cases: (i) *Native*, which denotes the case that only initiates the TEE OS kernel during each reboot; and (ii) MINITEE, which denotes the that all TA threads are initiated during each reboot.

We define the booting time as time interval from hardware power-on to MINITEE handover CPU to Normal World. In this test, PMU has been used since the GIC and the timers are not yet configured,

and the measurements have been performed 20 times for each case on Zedboard. As shown in Table 2, MINITEE requires more time to setting up the environment.

Table 2. MINITEE booting time.

	Booting Time	
	Worst-Case (ms)	Std
Native	31.14	0.13
MINITEE	83.35	0.25

Task worst case response time (WCRT): To study the run-time performance of MINITEE, we experimented with following cases: *Pure*, which denotes the original case that all sub-functions running in Normal World without using MINITEE functionalities; and aforementioned two cases (i.e., *Native* and MINITEE), and *OP-TEE*. We also added another dimension (i.e., whether the FreeRTOS's sysTick is enabled.) for each case, to study interrupt influence to tasks' WCRT.

We measured the WCRT for each task by separately running them on a FreeRTOS. A PMU-specific instruction was added at the beginning and end of task code to be measured. Results were gathered in clock cycles and converted to microseconds accordingly to the processor's frequency (667 MHz). The maximum value was selected from 500 collected samples as the WCRT for this task as described in Table 3. Note that, MMU, data and instruction cache and branch prediction were disabled on both Secure World and Normal World to minimize the interference as much as possible.

From Table 3, we can draw the following findings: (i) By comparing with the WCRT value measured in *Pure* case, our solution causes additional time overhead due the RPC protocol and world switching cost. Note that, MINITEE achieves a smaller overhead increase in MINITEE case than *Native*, as MINITEE resumes/suspends the corresponding *TEE task* rather than creates/destroys it. Also MINITEE can achieve a smaller variations. (ii) sysTick interrupt in Normal World could potentially increase the system overhead, since additional world switching overhead will be covered. (iii) Another interesting thing is that, the execution time of function Mission set-point decryption experiences a smaller increase than Image Encryption, because of the different share memory block size. The heavy performance overhead of OP-TEE mainly comes from the communication channel between Secure World and Normal World. Such communication channel slows down the invoked functions due to the introduction of two time-consuming mechanisms: connection maintenance to the Secure World (e.g., initialize/finalize context, open/close session), and invoking the partitioned functions in the Secure World (e.g., allocate/release shared memory, marshal and un-marshal parameters). MINITEE decreases such overhead by introducing the suspension/restoration of *TEE tasks*.

Table 3. Case study timing parameters.

Task Name	Pure		Native		MINITEE		OP-TEE	
	sysTick-dis	sysTick-en	sysTick-dis	sysTick-en	sysTick-dis	sysTick-en	sysTick-dis	sysTick-en
	wcrt(ms)	wcrt(ms)	wcrt(ms)	wcrt(ms)	wcrt(ms)	wcrt(ms)	wcrt(ms)	wcrt(ms)
Control	1.61	1.64	2.8	2.95	2.33	2.53	20.35	20.64
Camera	17.53	18.03	28.75	30.46	21.41	27.33	36.28	39.53
Network	0.035	0.044	0.036	0.044	0.035	0.044	0.035	0.44

6.3. Performance Evaluation

In this section, we detailed the overhead causing by each component of MiniTEE and assessed the interrupt latency in MiniTEE-enabled FreeRTOS.

A. Partition TA Invoking

To evaluate the TA invoking time we used the Performance Monitor Unit (PMU) component as described in previous section. Also, MMU, data and instruction cache and branch prediction were

disabled in both Secure World and Normal World. The values represent the average and the standard deviation of 2000 collected samples.

The list of internal activities to perform a full TA invoking are:

- (1) Initialize a MINITEE context through invoking MINITEE-Driver: Time since the real-time task calls API TEE_InitializeContext until MINITEE-Driver issues a world switching using SMC 0;
- (2) SMC from *Normal world* handling: Time since the processor enters in the monitor's vector table until restore the *Secure world* context;
- (3) Create a new *TEE task* or resume a previous suspended one: Time since MINITEE scheduler occupies the CPU until the called TA starts to execute on *TEE task's* stack;
- (4) Suspend MINITEE scheduler: Time since TA finishes its execution until MINITEE scheduler issues a SMC 0 to switch world back to Normal World;
- (5) Handle SMC from *Secure world* in Monitor module: Time since the processor enters in the monitor's vector table until restore the *Normal world* context ;
- (6) Finalize the MINITEE context: Time since the processor gets back to *Normal world* until MINITEE-Driver frees up the shared memory and goes back to the execution of the calling real-time task.

Table 4 presents the collected results. As it can be seen, the complete partition-switch operation takes around 131.2 μ s. This value is the time cost of a round-trip TA invoking, that must be considered when designing a RTOS. The most time-consuming part of a TA invoking is resume/suspend a *TEE task*. Because MINITEE will copy memory blocks between shared memory and the corresponding *TEE task* stack.

Table 4. Partitioned TA invoking time cost (in μ s).

World Switch	Operation	Performance (μ s)		Time Cost (μ s)
		\bar{x}	s	@667 MHz
Switch to S World	(1) Initialize MINITEE context	5.5	0.1	5.7
	(2) SMC from NS world handling	5.2	0.1	5.7
	(3) Create/resume TEE task	52.4	2.7	63.5
Switch to N World	(4) Suspend MINITEE OS scheduler	43.2	0.9	46.8
	(5) SMC from S world handling	3.9	0.1	4.8
	(6) Finalize MINITEE context	4.4	0.1	4.8

B. Interrupt Latency

Interrupt latency is the measurement of system's response time to an interrupt, which corresponds to the elapsed time between interrupt assertion and the instant when a response happens. Specifically, it is the summarization of following items. t_H is the hardware dependent time which depends on the interrupt controller on the hardware platform, as well as the type of the interrupt; t_{OS} is the OS-specific induced overhead; and t_{TEE} is the MINITEE-specific introduced overhead.

The results showed that the latency in the *Native* system (FreeRTOS) is 0.83 microseconds, which corresponds to the average interrupt handling overhead of our system, because when running in Normal World, the IRQ requests are directly handled by native FreeRTOS without introducing other system overhead. The t_{TEE} represents the extra overhead induced by our approach, which only occurs when the CPU running in Secure World. Since MINITEE runs with all interrupt sources disabled, the worst case scenario happens when an IRQ request (e.g., RTOS tick) arrives while a context switch from the Normal World to the Secure World is starting. In this case, the request is handled with a worst case interrupt latency of 2.53 microseconds. Note that, the overhead introduced on latency has a deterministic upper bound (2.53 microseconds), it should be taken into account when designing the real-time system.

7. Related Work

Security for real-time systems: Security for embedded hardware and software is becoming a main focus of recent works [23,24]. For real-time systems, the emphasis is usually on the trade-off between real-time constraints and security levels [25–33]. Mohan et al. [27,28] considered information leakage through storage timing channels using architectural resources (e.g., caches) shared between real-time tasks with different security levels. Their following work [29] introduced a schedule randomization protocol to minimize predictability of real-time system schedulers. All those works made the same assumption, which is that information with different security levels are sharing the same hardware resources, thereby causing side-channels. TrustZone-assisted TEEs can greatly reduce such possibilities thanks to the hardware-level extension of ARM TrustZone. Work by Hasan et al. [30,31] consider sporadic servers or similar approaches to add security related functionalities into the system. SecureCore in [32] utilizes a separate core on a multi-core platform to monitor the runtime behavior of real-time software on the other cores. Kim et al. [33] proposed a MPU-based memory isolation mechanism for real-time micro-controller systems through customized memory view switching. All those techniques provide the privilege for protection mechanisms via running them on OS-level or additional hardwares. While TEEs can be used in conjunction rather than instead of those techniques by placing them into the trusted environment to provide the strongest isolation.

TrustZone-assisted virtualization: There do exist some works to secure RTOS based on virtualization techniques [34–36]. For instance, Pinto et al. [34,35] presented virtualization solutions for both ARM v7-A and v8-M architecture which allow Linux OS and RTOS to run simultaneously while maintaining real-time performance. In FreeTEE [36], security services are treated as lower priority tasks of a RTOS running in the Secure World. All those works made the same assumption, which is that RTOS kernel and corresponding real-time applications are all trusted and have a higher security level than the Normal World applications. In the other direction, we believe that the Secure World should only be used to protect sensitive information. As such, only sections of task code/data are allowed to run inside TEE, thereby minimizing the size of TCB.

TrustZone-assisted TEEs: Most existing research on TrustZone-enabled TEE solutions target the mobile world [9]. One promising open-source TEE solution that can be integrated with RTOS is OP-TEE [13], which has been successfully ported to run alongside a real-time Linux environment [37]. However, the overhead associated with TEE and corresponding impacts on hard real-time deadlines are ignored. Moreover, since OP-TEE is initially designed for Linux, it will bring many constraints (e.g., memory size, driver libraries, etc.) when integrating with low-end RTOSs (e.g., FreeRTOS [38], NuttX [39], etc.).

Code refactoring to fit into TEEs: There is a line of research that employs program slicing techniques to automatically partition legacy applications into two parts to adapt TEE architecture [5,12,40]. For instance, RT-Trust [40] automatically separates the code/data for each real-time applications into two parts: one for the Secure World and the other for the Normal World, and then TEE-specific codes are patched to both parts with the consideration of real-time constraints. Our task model follows this logic, as we believe it is an efficient way to adapt existing real-time systems to use TEE solutions.

8. Conclusions

While trusted execution environments (TEEs) provide industry standard security and/or isolation, TEE requests through secure monitor calls (SMCs) attribute to large time overhead and weakened temporal predictability [18]. In this paper we present MINITEE to understand, evaluate and discuss the benefits and limitations when integrating TrustZone-assisted TEEs with low-end RTOSs. We demonstrate how MINITEE can be adequately exploited for meeting the real-time needs, while presenting a low performance cost on running alongside low-end RTOSs.

Author Contributions: Conceptualization, S.L. and N.G.; methodology, S.L.; software, S.L.; validation, S.L.; formal analysis, S.L.; investigation, S.L.; resources, S.L.; data curation, S.L.; writing—original draft preparation, S.L.; writing—review and editing, S.L. and N.G.; visualization, S.L.; supervision, N.G., Z.G. and W.Y.; project administration, N.G.; funding acquisition, N.G. and W.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This paper is sponsored by the National Natural Science Foundation of China (NSFC) under Grant 61672140 and the National Natural Science Foundation of China (NSFC) under Grant 61532007.

Acknowledgments: This paper is sponsored by the National Natural Science Foundation of China (NSFC) under Grant 61672140 and the National Natural Science Foundation of China (NSFC) under Grant 61532007.

Conflicts of Interest: No conflicts of interest.

References

1. Altawy, R.; Youssef, A.M. Security, privacy, and safety aspects of civilian drones: A survey. *ACM Trans. Cyber-Phys. Syst.* **2017**, *1*, 7. [CrossRef]
2. Lesi, V.; Jovanov, I.; Pajic, M. Network scheduling for secure cyber-physical systems. In Proceedings of the IEEE Real-Time Systems Symposium (RTSS), Paris, France, 5–8 December 2017.
3. Sabt, M.; Achemlal, M.; Bouabdallah, A. A. Trusted Execution Environment: What It is, and What It is Not. In Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA, Helsinki, Finland, 20–22 August 2015; Volume 1, pp. 57–64.
4. Arm. ARM Security Technology Building a Secure System using TrustZone Technology. Available online: https://static.docs.arm.com/genC009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf (accessed on 10 July 2020).
5. Rubinov, K.; Rosculete, L.; Mitra, T.; Roychoudhury, A. Automated partitioning of android applications for trusted execution environments. In Proceedings of the 2016 IEEE/ACM ICSE, Austin, TX, USA, 14–22 May 2016; Volume 1, pp. 923–934.
6. Challener, D.C.; Safford, D.R. Encrypted File System using TCPA. U.S. Patent 7,343,493, 10 July 2008.
7. Darwish, S.M.; Guirguis, S.K.; Zalat, M.S. Stealthy code obfuscation technique for software security. In Proceedings of the 2010 IEEE ICCES, Cairo, Egypt, 30 November–2 December 2010, Volume 1; pp. 923–930.
8. ARM. “TrustZone Technology for ARMv8-M Architecture, Version 2.1”. Available online: https://static.docs.arm.com/100690/0201/armv8_m_architecture_trustzone_technology_100690_0201_01_en.pdf (accessed on 1 October 2018).
9. Pinto, S.; Santos, N. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv. (CSUR)* **2019**, *51*, 130. [CrossRef]
10. GlobalPlatform. Available online: <https://globalplatform.org/> (accessed on 26 May 2020).
11. Technology, G.D.; Globalplatform. TEE Client API Specification. pp. 1–58. Available online: <https://globalplatform.org/specs-library/tee-client-api-specification/> (accessed on 10 July 2020).
12. Ye, M.; Sherman, J.; Srisa-An, W.; Wei, S. TZSlicer: Security-aware dynamic program slicing for hardware isolation. In Proceedings of the 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), Washington, DC, USA, 30 April–4 May 2018; pp. 17–24.
13. Linaro. OP-TEE. Available online: <https://wiki.linaro.org/WorkingGroups/Security/OP-TEE> (accessed on 26 May 2020).
14. Zhao, S.; Zhang, Q.; Qin, Y.; Feng, W.; Feng, D. SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 1723–1740.
15. McGillion, B.; Dettenborn, T.; Nyman, T.; Asokan, N. Open-TEE—An open virtual trusted execution environment. *TrustCom. arXiv* **2015**, arXiv:1506.07367v2. doi:10.1109/Trustcom.2015.400.
16. Mukherjee, A.; Mishra, T.; Chantem, T.; Fisher, N.; Gerdes, R. Optimized trusted execution for hard real-time applications on COTS processors. In Proceedings of the 27th International Conference on Real-Time Networks and Systems, Toulouse, France, 6–8 November 2019; pp. 50–60.
17. Liu, Y.; An, K.; Tilevich, E. RT-Trust: Automated refactoring for different trusted execution environments under real-time constraints. *J. Comput. Lang.* **2020**, *56*, 1–49. doi:10.1016/j.colan.2019.100939. [CrossRef]
18. Buttazzo, G.C.; Hard. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*; Springer: Berlin, Germany, 2011.

19. Cho, H.; Zhang, P.; Kim, D.; Park, J.; Lee, C.H.; Zhao, Z.; Doup, A.; Ahn, G.J. Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone. In Proceedings of the 34th Annual Computer Security Applications Conference, San Juan, PR, USA, 3–7 December 2018; pp. 441–452.
20. Santos, N.; Fonseca, P. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 18–20 May 2020.
21. Dong, P.; Burns, A.; Jiang, Z.; Liao, X. TZDKS: A new TrustZone-based dual-criticality system with balanced performance. In Processing of the 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Hakodate, Japan, 28–31 August 2018.
22. Shen, D. Exploiting TrustZone on android. *Black Hat USA* **2015**, *2*, 267–280.
23. Gai, K.; Qiu, L.; Chen, M.; Zhao, H.; Qiu, M. SA-EAST: Secur. -aware Effic. data Transm. ITS Mob. Heterog. cloud Comput. *TECS* **2017**, *16*, 60. [[CrossRef](#)]
24. Corteggiani, N.; Camurati, G.; Francillon, A. Inception: System-wide security testing of real-world embedded systems software. In Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 309–326.
25. Ma, Y.; Jiang, W.; Sang, N.; Zhang, X. Arcsm: A distributed feedback control mechanism for security-critical real-time system. In Proceedings of the 10th IEEE ISPA, Leganes, Madrid, Spain 10–13 July 2012; pp. 379–386.
26. Jiang, K.; Lifa, A.; Eles, P.; Peng, Z.; Jiang, W. Energy-aware design of secure multi-mode real-time embedded systems with FPGA co-processors. In Proceedings of the 21st ACM RTNS. Sophia Antipolis, France, 16–18 October 2013; pp. 109–118.
27. Mohan, S.; Yoon, M.K.; Pellizzoni, R.; Bobba, R. Real-time systems security through scheduler constraints. In Proceedings of the 26th IEEE ECRTS, Madrid, Spain, 8–11 July 2014; pp. 129–140.
28. Pellizzoni, R.; Paryab, N.; Yoon, M.K.; Bak, S.; Mohan, S.; Bobba, R.B. A generalized model for preventing information leakage in hard real-time systems. In Proceedings of the 21st IEEE RTAS, Seattle, WA, USA, 13–16 April 2015; pp. 271–282.
29. Yoon, M.K.; Mohan, S.; Chen, C.Y.; Sha, L. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In Proceedings of the 22nd IEEE RTAS, Vienna, Austria, 11–14 April 2016; pp. 1–12.
30. Hasan, M.; Mohan, S.; Bobba, R.B.; Pellizzoni, R. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In Proceedings of the IEEE RTSS 2016, Porto, Portugal, 29 November–2 December, 2016; pp. 123–134.
31. Hasan, M.; Mohan, S.; Pellizzoni, R.; Bobba, R.B. A design-space exploration for allocating security tasks in multicore real-time systems. In Proceedings of the IEEE DATE, Dresden, Germany, 19–23 March 2018; pp. 225–230.
32. Yoon, M.K.; Mohan, S.; Choi, J.; Kim, J.E.; Sha, L. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In Proceedings of the 19th IEEE RTAS, Philadelphia, PA, USA, 9–11 April 2013; pp. 21–32.
33. Kim, C.H.; Kim, T.; Choi, H.; Gu, Z.; Lee, B.; Zhang, X.; Xu, D. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In Proceedings of the NDSS 2018, San Diego, CA, USA, 18–21 February 2018; pp. 31–42.
34. Pinto, S.; Pereira, J.; Gomes, T.; Tavares, A.; Cabral, J. LTZVisor: TrustZone is the key. In Proceedings of the 29th IEEE ECRTS, Dubrovnik, Croatia, 27–30 June 2017; pp. 137–158.
35. Pinto, S.; Araujo, H.; Oliveira, D.; Martins, J.; Tavares, A. Virtualization on TrustZone-Enabled Microcontrollers? Voilà!, In Proceedings of the 25th IEEE RTAS, Montreal, QC, Canada, 16–18 April 2019; pp. 293–304.
36. Pinto, S.; Oliveira, D.; Pereira, J.; Cabral, J.; Tavares, A. FreeTEE: When real-time and security meet. In Proceedings of the 20th IEEE ETFA, Luxembourg, 8–11 September 2015; pp. 1–4.
37. Liu, R.; Srivastava, M. PROTC: PROTeCting drone’s peripherals through ARM trustzone. In proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications, Niagara Falls, NY, USA, 23 June 2017; pp. 1–6.
38. FreeRTOS. “The FreeRTOS Real-Time Operating System”. Available online: <https://www.freertos.org> (accessed on 10 July 2020).

39. NuttX. “NuttX Real-Time Operating System”. Available online: <https://nuttx.apache.org/> (accessed on 10 July 2020).
40. Liu, Y.; An, K.; Tilevich, E. RT-trust: automated refactoring for trusted execution under real-time constraints. In Proceedings of the 17th GPCE, Boston, MA, USA, 5–6 November 2018; pp. 175–187.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).