Optimized Trusted Execution for Hard Real-Time Applications on COTS Processors

Anway Mukherjee[†], Tanmaya Mishra[†], Thidapat Chantem[†], Nathan Fisher[‡], Ryan Gerdes[†]
[†]Department of Electrical and Computer Engineering, Virginia Tech, USA.
Email: {anwaym, tanmayam, tchantem, rgerdes} @vt.edu
[‡]Department of Computer Science, Wayne State University, USA.
Email: fishern@wayne.edu

ABSTRACT

While trusted execution environments (TEE) provide industry standard security and isolation, its implementation through secure monitor calls (SMC) attribute to large time overhead and weakened temporal predictability, potentially prohibiting the use of TEE in hard real-time systems. We propose super-TEEs, where multiple trusted execution sections are fused together to amortize TEE execution overhead and improve predictability through minimized I/O traffic and reduced switching between normal mode and TEE mode of execution. Super-TEEs may, however, violate a task's timing requirement and impact the schedulability of the system. We present a technique to enforce the correct timing requirement of a task, along with a sufficient test for schedulability in uniprocessors. We also, discuss CT-RM, a static task assignment and partitioned scheduling algorithm to schedule super-TEEs, alongside other real-time tasks, on multicore systems. Experimental results on a Raspberry Pi 3B, further confirmed by simulations, show that CT-RM outperforms the state-of-the-art technique in terms of usable utilization by 12% on average and up to 27%.

1 INTRODUCTION

As real-time embedded systems become more complex and interconnected, certain sections or parts of the systems may need to be secured to prevent unauthorized access, or isolated to ensure correctness. In the case of hard real-time systems such as a surveillance unmanned aerial vehicle (UAV), there is a need to provide security while maintaining strict real-time deadlines. That is, the UAV may collect confidential information such as surveillance co-ordinates. An attacker could impersonate ground control and easily extract such sensitive information from the system. In such cases, we must isolate sensitive information from the rest of the system to prevent such inadvertent leaks to an unauthorized party.

Trusted execution environments (TEE) [1] are a widely used solution for industry standard platform-level security in embedded systems. TEE provides virtualization for a secure execution environment by leveraging architecture-specific hardware security extensions to protect and isolate information from access by a third party [2, 3]. The wide spread availability of TEE in commercial-off-the-shelf (COTS) systems (1) dramatically reduces the need for the costly development and manufacturing of custom hardware solutions such as hardware-based encryption, code obfuscation, etc. [4, 5], and, (2) allows for quick redeployment as platform virtualization on top of ARM TrustZone enabled security extensions. However, it may be challenging to adopt TEE in hard real-time systems, as TEE can incur large time overhead and highly variable execution times [6]. Specifically, each instance of TEE execution (in ARM TrustZone) is initiated by a setup phase and exits through a destroy phase. Since TEE leverages architecture-specific secure monitor calls (SMC) to realize these phases, the time overhead associated with TEE execution, including setting up and tearing down a TEE session, requires $18{,}500\,\mu s$ on a Raspberry Pi 3B (Rpi 3B) (Table 1). Moreover, data/instruction fetches and writebacks during TEE execution is a major cause for large variation in task execution times, thus weakening predictability.

In this work, we focus on TEE supported by ARM TrustZone, as the latter is popular in smartphones and other embedded mobile devices [7], while Intel SGX primarily targets resource-rich servers [8]. Our objective is to reduce the overall number of SMCs in applications where multiple sections of the code that must run in TEE are fused together to (1) amortize the time overhead associated with SMCs, (2) minimize the associated I/O traffic (memory fetches and writebacks) since they are the primary source of variable time overhead, and, (3) improve the temporal predictability of the system by reducing the number of switching between secure and non-secure environments due to SMCs. We also present a task assignment and scheduling framework for real-time trusted execution on readily available multicore systems. Since our work focuses on optimizing trusted executions for use in hard real-time systems without changing the underlying TEE implementation, we do not examine the security benefits and drawbacks of TEE. Instead, readers are referred to existing work [9–11]. Our main contributions are:

(1) We present our approach for reducing TEE overhead by forming super-TEEs, which involve fusing together two or more secure code or application sections that require TEE. We describe a task model to represent a super-TEE, discuss a technique to enforce timing correctness, and derive a sufficient condition for schedulability for super-TEEs and other real-time tasks on uniprocessor systems under a fixed priority scheduling scheme.
(2) We propose a fixed-priority task-to-core assignment and partitioned scheduling technique called CT-RM for multicore systems. We show that our approach never performs worse, and in fact often outperforms the widely used first-fit partitioned rate-monotonic

RTNS 2019, November 6–8, 2019, Toulouse, France
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7223-7/19/11…$15.00
https://doi.org/10.1145/3356401.3356419

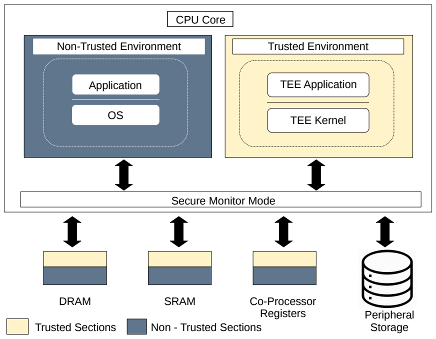


Figure 1: Architecture-specific platform security extensions for TEE in CPU, memory subsystem and peripheral storage.

scheduling algorithm for multicore systems. We validate our approach and assess its performance in a custom-built simulated environment.

(3) We validate our approach on a Raspberry Pi running Linux (with `preempt-RT` patches for real-time behavior) and OP-TEE, an open-source TEE implementation.

2 PRELIMINARIES

2.1 ARM TrustZone

The ARM TrustZone [12] is an embedded and secure virtualization platform for COTS embedded systems. Platform virtualization enables enhanced system security through either (1) TEE or, (2) trusted platform module (TPM). TEE consists of a virtualized environment where code that requires trusted execution can be securely executed in isolation from the entire system. Figure 1 shows two separate execution environments, namely normal world (running the non-trusted OS) and secure world (running the trusted OS). The trusted OS, in comparison to non-trusted OS, has limited hardware/software features, and reduced functionality. The TEE and non-TEE cannot simultaneously execute on same processor core since the processor can be in one mode only at any given time, but can run in parallel on separate cores. Communication between TEE and non-TEE environments during a mode switch is established through an architecture-specific secure message passing protocol.

In non-trusted execution environment, the non-secure applications, i.e., those that do not require TEE execution, run on standard embedded hardware resources. In a trusted environment, all trusted execution code and its data are stored in, copied to, and executed in isolation on specially augmented secure hardware resources, e.g., CPU, memory, and peripherals. These secure hardware extensions provide data encryption to protect secure environment data/code from being accessed by the non-trusted software. The trusted OS may also logically deactivate a subset of existing secure peripherals in respective execution environments in order to minimize the probability of unauthorized access on unprotected peripherals. TrustZone software delimits the code running in normal world from changing the secure OS system state. This means that if an attack is routed through the normal OS, it is confined to the access privileges of only the non-secure environment. On the other hand, trusted OS is vulnerable to internal attacks. A security breach can happen through the secure OS if a trusted application chosen to run on TEE is inherently

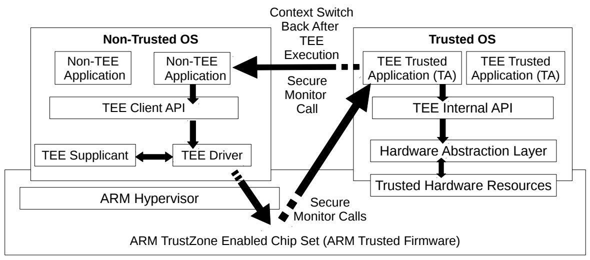

Context Switch Back After TEE Execution
Non-Trusted OS
Non-TEE Application
Non-TEE Application
TEE Client API
TEE Supplicant
TEE Driver
Secure Monitor Call
Trusted OS
TEE Trusted Application (TA)
TEE Trusted Application (TA)
TEE Internal API
Hardware Abstraction Layer
Trusted Hardware Resources
ARM Hypervisor
Secure Monitor Calls
ARM TrustZone Enabled Chip Set (ARM Trusted Firmware)

Figure 2: Design framework of the ARM TrustZone software stack showing the flow of execution and data exchange between the secure and non-secure execution environments.

malicious. Hence, it is the programmer's responsibility to ensure that the trusted applications do not inadvertently introduce any security vulnerabilities.

2.2 Open Portable Trusted Execution Environment (OP-TEE)

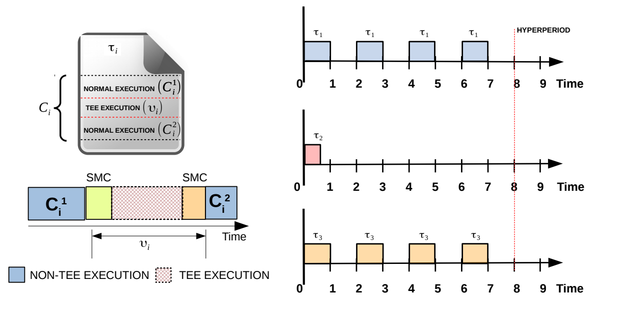
OP-TEE [13] is an open-source TEE implementation by Linaro to integrate ARM-compatible standard Linux with ARM TrustZone. It uses the standardized GlobalPlatform TEE specifications [14] to construct a framework for ARM TrustZone to co-exist with a standard Linux distribution. Figure 2 shows the detailed setup of an OP-TEE port into a Linux environment. The OP-TEE OS consists of three main components, (1) a hardware abstraction layer which forms the backbone of platform virtualization and provides a communication channel with the non-trusted OS, (2) a minimal secure kernel and its associated TEE internal APIs to support trusted execution and (3) a set of trusted applications which can run in isolation on OP-TEE OS. Similarly, the co-existent non-trusted OS is augmented with a secure interface to communicate with a trusted application.

Non-secure applications use the TEE-client APIs exposed by the OP-TEE driver implemented in the Linux kernel. When a non-secure application requests for TEE, the driver intercepts the SMC, suspends the calling application and triggers an SMC handler in the OP-TEE OS. The OS then loads the required TA in a secure kernel thread, executes the TA and switches back control to the non-trusted OS. When an interrupt comes from either secure or non-secure world, OP-TEE OS saves the TA context and suspends it, triggers the interrupt handler (or switches to normal world and triggers the handler there if it is a non-secure interrupt), reloads the TA context and continues execution. Note that a TA need not continue its execution on the same core after reload.

The OP-TEE OS implementation does not have a process scheduler. Hence, it uses the process scheduler of the host OS to run its secure kernel thread. However, OP-TEE OS maintains an active secure thread stack to service non-secure interrupts, thread migration and premature process termination. Note that the trusted OS reserves the right to define and service its security specific implementations. For more details regarding OP-TEE, readers are referred to the GlobalPlatform specifications [14] and OP-TEE project website [13].

2.3 System Model

We assume a set of homogeneous cores $\mathbf{P}$, where each core $p_j \in \mathbf{P}$, can switch between secure (TEE) and non-secure (non-trusted OS)



(a) (b)

Figure 3: (a) Our real-time task model. For a task τ_i without TEE requirement, $\upsilon_i = C_i^2 = 0$; (b) Example offline profiling of the task set (Table 7) for super-TEE construction.

mode of operation using SMCs. While the non-secure mode is used for executing normal real-time tasks, the secure mode runs TEE sections of code. In addition, we consider a set of n independent synchronous periodic hard real-time tasks, which have been re-factored to isolate TEE sections from its non-secure execution segments [6]. Therefore, each task τ_i, $i = 1, \ldots, n$, can be described by the following tuple: $(T_i, C_i^1, \upsilon_i, C_i^2)$, where C_i^1 and C_i^2 are the worst-case execution times (WCETs) in non-secure mode, and υ_i is the WCET in TEE mode, and T_i is the period of execution. We assume that deadlines are equal to periods. Figure 3(a) depicts our real-time task model. For a task with TEE requirements, two normal computation segments (C_i^q $q = 1, 2$) are interleaved by a single trusted execution segment[1] (υ_i). Similarly, for a task τ_i that does not require TEE, $\upsilon_i = C_i^2 = 0$. The upper bound on the combined WCET of a task τ_i is given by $C_i = \sum_{q=1}^{2} C_i^q + \upsilon_i$.

We consider partitioned scheduling in this work and will leverage the first-fit rate-monotonic (RM-FF) scheduling policy (Section 5). A sufficient utilization based test for RM for a uniprocessor is restated below. The utilization U_i of task τ_i is defined as $\frac{C_i}{T_i}$, and the utilization of a task set (U_T) is the sum of the utilization of all the tasks, n, in the task set, i.e., $\sum_{i=1}^{n} U_i$. Since we target multicore systems in this work, the Liu-layland (LL) limit [15], given by Equation 1, is used to perform per-core schedulability test when task to core assignment is carried out.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1). \tag{1}$$

Note that any bound can be adapted for use in our framework. However, we choose the LL limit since it has low computational complexity and can easily be calculated based on the number of tasks [16]. Other schedulability tests can be used, and is left for future work.

2.4 Creating Trusted Execution Segments

A TEE section delimits the private and/or sensitive section of the source code (Figure 4). We list the process of partitioning a task

[1]Though we consider a single trusted execution interval for simplicity, it is not advisable to have multiple TEE execution sections interspersed with non-secure computation segments within a TEE task, since each trusted execution has high time overhead (Table 1).

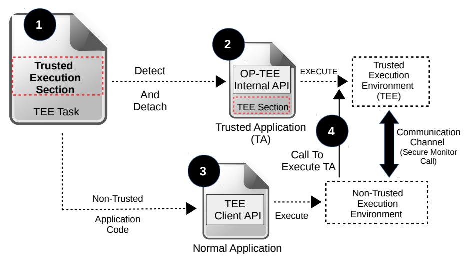

1
Trusted Execution Section
TEE Task
Detect
And Detach
2
OP-TEE Internal API
TEE Section
Trusted Application (TA)
EXECUTE
Trusted Execution Environment (TEE)
4
Call To Execute TA
Communication Channel (Secure Monitor Call)
3
TEE Client API
Normal Application
Execute
Non-Trusted Execution Environment
Non-Trusted Application Code

Figure 4: Step-by-step example showing how to convert a software program into separate executable.

Table 1: TEE overhead on RPi 3b running OP-TEE

API name	Latency in us
`TEEC_InitializeContext()`	200
`TEEC_OpenSession()`	17000
`TA_InvokeCommand()`	250
`TEEC_CloseSession()`	1200
`TEEC_FinalizeContext()`	100

with TEE requirements into a secure and non-secure executable. We use an existing work [6] to automatically partition the original code around a segment which is annotated by the software developer to demarcate for TEE computation into a non-trusted executable (step 3), and a trusted application (step 2) as shown in Figure 4. Step 1 takes a task τ_i with trusted execution requirements, identifies the annotated TEE section (υ_i), and detaches it from the original source code. In step 2, we take the detached TEE segment (υ_i), instrument it with TEE-specific internal APIs, and create a separate executable known as trusted application (TA). Similarly, in step 3, we instrument the non-trusted section(s) of the code (C_i^q $q = 1, 2$) with TEE-specific interfaces to link it to the TA, and create a separate normal non-secure executable. Finally, the two executable form a sequential flow of execution wherein the non-secure executable in the non-trusted OS starts its execution followed by a switch to the TEE to run the TA in step 4. Ultimately, TA execution returns the context back to non-secure executable to complete further execution.

3 MOTIVATION

While TEE provides industry standard certification, security and isolation, its use in hard real-time systems is challenging. As previously reported [6], and verified through experimentation (Table 1), the time overhead associated with SMCs is large. The increase in WCETs for TEE execution can be primarily[2] attributed to architecture-specific SMCs to setup and destroy a trusted environment for secure execution. Note that while this increase in tasks' WCETs can be considered during schedulability analysis, many systems are resource constrained. In addition, it is not feasible to always keep TEE sessions open since a single TEE session is limited by the size of its secure private cache (4KB for our board). Said cache is used to store and execute the trusted executable, along with any other required data. Hence, the number of applications/trusted segments that can execute in a single session is limited. Even if the size of the secure private cache is large, always keeping TEE sessions open would

[2]Running a TEE section on a processor requires an exclusive access for the trusted OS to execute the trusted segment on that particular core (i.e., the core must be in secure mode and not normal mode). While interrupts exist to switch back to the non-trusted environment and service normal execution, a mode change adds to the time overhead associated with trusted execution.

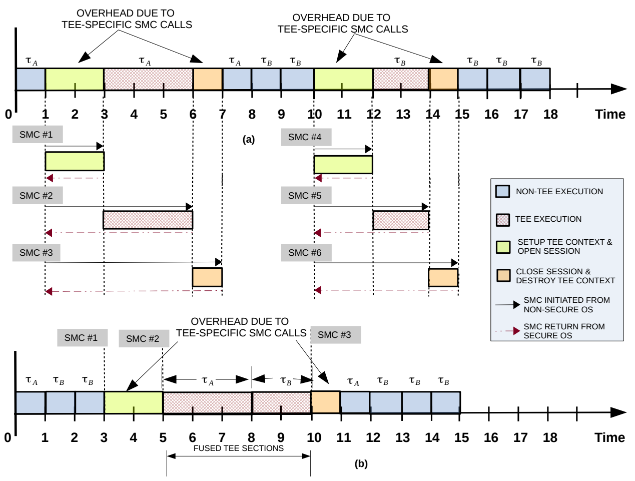

OVERHEAD DUE TO TEE-SPECIFIC SMC CALLS
OVERHEAD DUE TO TEE-SPECIFIC SMC CALLS
τ_A
τ_B
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 Time
(a)
SMC #1
SMC #2
SMC #3
SMC #4
SMC #5
SMC #6
NON-TEE EXECUTION
TEE EXECUTION
SETUP TEE CONTEXT & OPEN SESSION
CLOSE SESSION & DESTROY TEE CONTEXT
SMC INITIATED FROM NON-SECURE OS
SMC RETURN FROM SECURE OS
FUSED TEE SECTIONS
(b)

Figure 5: A motivating example where (a) two tasks need six SMC calls to access two separate instances of TEE execution, but, (b) fused TEE execution sections reduce the number of SMC calls to three.

Table 2: Task Set 1

Task (τ)	C_i^1	v_i^s	v_i^t	v_i^d	$v_i = v_i^s + v_i^t + v_i^d$	C_i^2	$C = \sum_{q=1}^{2} C_i^q + v_i$	T
τ_A	1	2	3	1	6	1	8	16
τ_B	2	2	2	1	5	3	10	17

mean dedicating a fixed number of cores to running TEE, resulting in a less efficient use of computing resources since these cores may often be idle. In addition, no intra-task communications are permitted during TEE execution by virtue of TEE security. For trusted segments requiring data for trusted computation, a new TEE session has to be open to load such data into the secure private cache, even if such inputs were generated from other trusted segments. In this work, we propose to amortize the time overhead associated with each call for secure execution in TEE by grouping two or more tasks which have TEE sections and fuse their trusted execution sections together into a super-TEE (Section 4).

We now provide a simple example to show how forming a super-TEE can help to decrease tasks' collective worst-case execution time (WCET). In this example we will ignore task deadlines for clarity. Consider two tasks τ_A and τ_B shown in Table 2. Let us assume that both tasks contain sections that require TEE execution. The upper bound on the non-trusted segment for both tasks τ_A and τ_B, given by $\sum_{q=1}^{2} C_i^q$, is equal to 2 and 5 respectively. Similarly, the upper bound on the trusted execution section, given by $v_i = v_i^s + v_i^t + v_i^d$, is equal to 6 and 5 respectively, where SMC_setup is denoted by v_i^s, SMC_destroy by v_i^d and the actual trusted computation duration inside TEE by v_i^t. Hence, C_A and C_B equal 8 and 10 time units, respectively. Figure 5(a) shows a possible schedule if we were to execute τ_A and τ_B separately. Figure 5(b) shows how the collective WCET of τ_A and τ_B, now 15 time units since $v_{AB} = 2 + 3 + 2 + 1 = 8$, is reduced when a super-TEE is formed. Clearly, reducing WCETs can help to improve schedulability. However, while fusing the TEE execution sections of tasks results in improved collective WCET, such a modification may change the collective period of the tasks under consideration, thereby, potentially violating the timing requirements of the tasks, and increasing the total system utilization. We address these challenges next.

4 OFFLINE SUPER-TEE CONSTRUCTION

We construct super-TEEs to amortize TEE execution overhead while maintaining logical and timing correctness. A *super-TEE* is defined as a real-time task consisting of a single secure section or code that require TEE execution, formed by fusing together the trusted execution sections of two or more real-time tasks. Since secure memory locations are allocated contiguously in TEE [13], we exploit the high spatial locality for secure memory locations to amortize the overhead associated with I/O transactions during trusted execution. We first introduce super-TEE and its construction while maintaining its logical correctness, followed by its representation using a real-time task model, and discuss its timing correctness. Finally, we present a technique to select which tasks to form super-TEEs to maximize the schedulability of the system.

4.1 Task Partitioning and TEE Fusions

The process of creating super-TEEs is carried out once offline. In this section, we discuss how to fuse TEE sections together. The question on which actual TEE sections to fuse together will be addressed in section 4.5. For the sake of simplicity, we consider fusing the TEE sections of two tasks in this paper. However, our approach can be applied to an arbitrary number of tasks with harmonic periods [17]. Section 2.4 explains the process of partitioning tasks with TEE requirements into separate executables. The v_i section forms a separate trusted executable (TA) from the C_i^q $q = 1, 2$ sections which is collectively denoted by a client application (CA). The TA is invoked by the CA through SMCs (Figure 4).

We focus on independent tasks for super-TEE construction in this work. Before fusing the TEE sections, each task τ_i is profiled for its real-time periodic behavior. That is, each task is run in isolation on a uniprocessor to determine when it would execute. We then collect this information over a hyperperiod (Figure 3(b)). Then, an existing framework [18] is leveraged to calculate the overlapping execution sections for each task tuple, and the data collected is stored as entries in Table 8. We then ascertain whether the TA is independent by checking the memory page access (using an existing automated tool [6]) and record the memory footprint of each TA. Since the maximum available private secure cache size in TEE is limited by its architecture-specific design constraint, the task-specific secure cache footprint delimits the number of TEEs that can be fused together so that super-TEE does not exceed the secure cache limit. We merge both TA-specific code/data into a single executable and augment the fused TA to pre-fetch the memory address blocks of the TAs of the fused tasks (example below). Merging multiple TEE sections into a single TA allows it to be loaded once, followed by multiple function calls, one for each of the TEE sections resulting in sequential execution. The trusted OS maintains the context of the secure execution (including instructions and meta-data not visible to the non-trusted OS). Our approach avoids the intermediate steps of unpredictable and time-consuming I/O transactions (memory writeback for the current TEE section of a task carried out while performing memory fetches for the subsequent TEE execution of the other task), leading to an improved overall temporal predictability. We also merge the CAs into a single executable while maintaining the overall temporal correctness of the tasks (see Section 4.3). To regulate our assumption, we ensure the isolation of data/instructions in the non-secure segments of the super-TEE during the offline profiling step. More importantly, the software developer must utilize an existing automated approach [6] to maintain the separation of resource between two CAs. The fused CA and its corresponding fused TAs form a super-TEE.

Figure 6 illustrate an example code transformation [6] to construct a super-TEE. Lines 1 – 19 show two independent tasks τ_1 and τ_2, both of which require TEE execution. In task τ_1, `funcA()` and `funcC()` form non-trusted execution sections, while `funcB()` requires TEE execution. Similarly, for τ_2, `funcX()` and `funcZ()` constitute the non-secure execution sections, while `funcY()` requires TEE execution. Lines 21 – 36 show re-factored code (CA) where tasks τ_1 and τ_2 are fused to form super-TEE while maintaining its logical correctness. Finally, Lines 38 – 45 show the corresponding trusted segments of tasks τ_1 and τ_2 which have been partitioned into single TAs for TEE execution. The Super-TEE calls TAs using TEE client APIs as demonstrated in lines 29 – 30 (Figure 6).

```
1  main(){ //Task τ1
2  funcA(); //Non−trusted code
3  funcB(); //Call funcB()
4  ...
5  funcC();//Rest of non−trusted code
6  }
7  //Annotated Segment: assume requires TEE execution
8  int funcB(){
9  ... } //Segment END
10
11 main(){ //Task τ2
12 funcX(); //Non−trusted code
13 funcY(); //Call funcY()
14 ...
15 funcZ();//Rest of non−trusted code
16 }
17 //Annotated Segment: assume requires TEE execution
18 int funcY(){
19 ... } //Segment END
20
21 //Super-TEE
22 main(){
23 funcA(); //Non−trusted code: Task τ1
24 funcX(); //Non−trusted code: Task τ2
25 //Start TEE execution using TEE client APIs
26 TEEC_InitializeContext(...);//Set up TEE context
27 TEEC_OpenSession(...); //TEE entry
28 //Call trusted applications (TAs)
29 TEEC_InvokeCommand(funcB,...);//TEE call: Task τ1
30 TEEC_InvokeCommand(funcY,...);//TEE call: Task τ2
31 TEEC_CloseSession(&sess);
32 TEEC_FinalizeContext(&ctx); //End TEE execution
33 ...
34 funcC()//Rest of non−trusted code: Task τ1
35 funcZ()//Rest of non−trusted code: Task τ2
36 }
37
38 //TA code: run in trusted execution environment
39 static TEE_Result funcB(...){
40 ... //funcB() body
41 return TEE_SUCCESS;}
42 ...
43 static TEE_Result funcY(...){
44 ... //funcY() body
45 return TEE_SUCCESS;}
```

Figure 6: Code snippet of two example tasks τ_1 and τ_2 which need to be re-factored for TEE execution, and fused into super-TEE **and its corresponding trusted segments** (TAs)**.**

4.2 Security Impacts

We make the observation that TEE overhead can be reduced with minimal (possibly no) security impact on trusted execution since our approach does not require modifications to TEE. Specifically, each trusted segment executes within a secure isolated context (TEE), which is implemented and maintained within the trusted OS. All instructions and meta-data specific to its execution is contained within its own secure thread in the trusted OS. Since each TEE execution is initiated from the non-trusted side, it requires a switch back from the trusted environment. In such a case, the trusted OS (i) maintains the context of the secure execution (not visible to the non-trusted OS) before (ii) resetting the execution environment while transferring the control over to the non-secure side. Constructing a super-TEE by fusing two separate trusted execution segments, thus, simply implies that both will run inside the same trusted environment, however, on isolated secure threads. While this ensures that an attacker in control of a task that includes TEE execution cannot subvert either the original trusted execution sections or the fused ones it, nonetheless, potentially allows for side channel attacks [19, 20] in some task instances (e.g., for fused tasks that accept attacker-controlled input) because the context (including registers and caches) is not securely removed between TEE sections.

For example, consider that two tasks τ_A and τ_B require TEE execution, and are fused to form a super-TEE task. Assume that an attacker can control the input to the non-TEE client application (CA) executable of task τ_B, which runs directly after task τ_A, which itself has sensitive data stored in the cache. Furthermore, assume that τ_B performs computations on the input data that results in cache access and that the attacker understands how this occurs (i.e., attacker has access to tasks τ_A and τ_B's code). If the compromised non-TEE executable (CA) the attacker controls allows the attacker to make timing measurements, it may be possible for the attacker to recover data from the cache [19, 20] based on how long computations take for task τ_B.

A possible solution to mitigate this threat would require flushing of cache (and clearing of CPU registers) between execution of two separate TEE sections for tasks τ_A and τ_B. Experimental results on our hardware test bed (Rpi 3B) shows an overhead of 22 μs to flush out up to 4 KB of secure cache data.

4.3 Super-TEE Task Model

While discussing the construction of a super-TEE in the previous section, we focused on the logical correctness of the tasks. We now examine the timing aspects of a super-TEE in this section. Let us revisit our motivating example in Section 3 where the corresponding synchronous task set is shown in Table 2. Let us consider fusing τ_A and τ_B together to form a super-TEE. Since τ_A and τ_B have different periods, there are two challenges associated with upholding temporal correctness: (1) executing the super-TEE with τ_B's period (17 time units) may be logically incorrect since τ_A (with smaller period) needs to run more frequently, and, (2) executing the super-TEE with

Table 3: Task Set 2

Task (τ)	C	T
τ_A	1	3
τ_B	1	7

Table 4: Modified Task Set 2

Task (τ)	C^{peak}	C^{nml}	T	l
τ_{AB}	1.7	1	3	2

τ_A's period (16 time units) may result in unnecessary resource usage since τ_B (with larger period) needs to run less frequently. Therefore, the challenge is to ascertain the time intervals where both τ_A and τ_B execute so that we can run the super-TEE, while at other times we run only τ_A since it is the task with a lower period. We model a super-TEE as a variant of the multi-frame task model [21]. A super-TEE task τ_{ij} constructed by fusing synchronous tasks τ_i and τ_j is defined as $(T_{ij}, C_{ij}^{peak}, C_{ij}^{nml}, l_{ij})$. T_{ij}, defined as the period of the super-TEE task τ_{ij}, is set as $\min\{T_i, T_j\}$. In our example task set (Table 2), $T_{AB} = 16$, since the period of task τ_A is shorter than task τ_B. The execution time parameter C_{ij}^{peak} corresponds to the WCET of a frame when the fused TEE section is running, and is calculated using the equation 2. Similarly, C_{ij}^{nml} corresponds to the WCET of a frame when the fused TEE section is not running, and is calculated using the equation 3. From Equations 2–3, we know that for each job instance of the task-tuple $\{\tau_A, \tau_B\}$ which coincides with the fused TEE section runs with a WCET of $C_{AB}^{peak} = 15$ time units, as discussed in Section 3, while the job instance which corresponds to non-fused TEE execution has a WCET of $C_{AB}^{nml} = 8$ time units , i.e., the WCET of τ_A, the task with the smaller period.

$$C_{ij}^{peak} = \sum_{q=1}^{2}\left[C_i^q + C_j^q\right] + v_i + v_j - (v_j^s + v_j^d). \tag{2}$$

$$C_{ij}^{nml} = C_i = \sum_{q=1}^{2} C_i^q + v_i. \tag{3}$$

$$l_{ij} = \left\lfloor \frac{T_j}{T_{ij}} \right\rfloor \tag{4}$$

The parameter l_{ij} captures the minimum inter-peak frame distance such that every l_{ij} consecutive frames contain at most *one* peak frame. In our example, since at every 16 time units interval, there must be at most *one* C^{peak} frame to account for each job instance of tasks τ_A and τ_B, we set $l_{AB} = 1$ (Equation 4). Tasks which are not super-TEEs have $C^{peak} = C^{nml}$. Now, for each task $\tau_i = (T_i, C_i^{peak}, C_i^{nml}, l_i)$ in the task set $\Psi = \{\tau_i : i = 1, \ldots, n\}$ the total task set utilization is given by

$$U_T = \sum_{i=1}^{n}\left[\frac{C_i^{nml}}{T_i} + \frac{C_i^{peak} - C_i^{nml}}{l_i T_i}\right] \tag{5}$$

The revised real-time parameters of the tasks in the task set (Table 2) is represented by the tuple $(16, 15, 8, 1)$, where $T_{AB} = 16$, $C_{AB}^{peak} = 15$, $C_{AB}^{nml} = 8$, and $l_{AB} = 1$.

4.3.1 Example #1: Let us consider two tasks with co-prime time periods. The real-time parameters of the tasks in this task set is given in Table 3. We will now help the readers walk-through the process of generating real-time parameters for this task τ_{AB}. We consider that each task has trusted segments that need to run in TEE. The upper bound on worst-case execution time ($\sum_{q=1}^{2} C_i^q + v_i$) for both

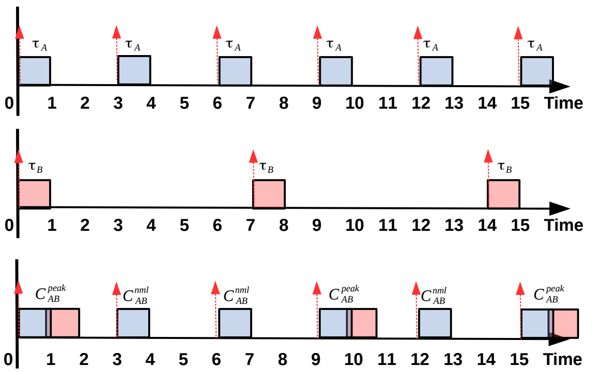


Figure 7: A example execution instant of the super-TEE (Table 4).

Table 5: Task Set 3

Task (τ)	C	T
τ_A	1	3
τ_B	1	4

Table 6: Modified Task Set 3

Task (τ)	C^{peak}	C^{nml}	T	l
τ_{AB}	1.7	1	3	1

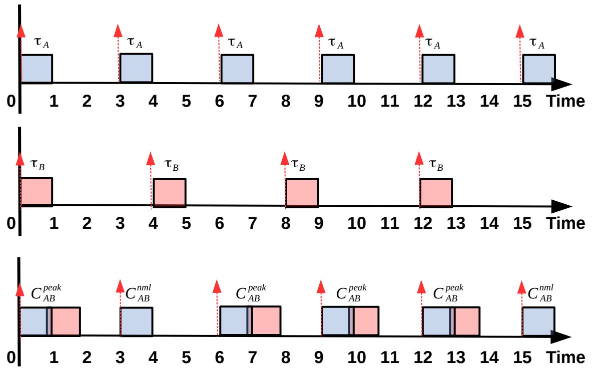


Figure 8: A example execution instant of the super-TEE (Table 6).

tasks τ_A and τ_B is set at 1. Also, let us assume that the total SMC overhead (v_i^s and v_i^d) for both tasks is 0.3 time units. Therefore, using Equations 2–3, we have $C_{AB}^{peak} = 1.7$ and $C_{AB}^{nml} = 1$. Similarly, T_{AB} is set to 3 ($\min\{T_A, T_B\}$). From Equation 4, we know that l_{AB} equals to 2, and super-TEE maintains temporal correctness as long as each $(T_{AB} \cdot l_{AB})$ time interval contains at most *one* C^{peak} frame (Figure 7). Note that at time 0, a job of τ_A and τ_B has each been released and so the super-TEE can execute its C_{AB}^{peak} frame (fused TEE section). In contrast, at time 6, the super-TEE will have to run its C_{AB}^{nml} frame since a new job instance of τ_B is yet to be released. Hence, the next C_{AB}^{peak} frame can only be formed at time 9. The same pattern is repeated after every hyperperiod of τ_{AB} (the least common multiple of the periods of τ_A, τ_B). The modified task set is shown in Table 4.

4.4 Feasibility of Super-TEE Candidates

The main objective for fusing TEE execution sections together is to save CPU cycles every time an instance of a super-TEE is scheduled as shown in Figure 5. While fusing the TEE execution sections of tasks reduce the collective WCET, such a modification may change the collective period of the tasks under consideration, thereby, increasing the total system utilization. We present a schedulability analysis to determine whether a super-TEE candidate is in fact feasible, i.e, the super-TEE candidate will not cause an increase in the total system utilization. A task-tuple, i.e., super-TEE candidate, that fails to meet the condition stated in Theorem 1 is removed from further consideration. That is, the tasks will execute independently without having their TEE sections fused as shown in Algorithm 2.

Observe that in Example #1 (Section 4.3.1), even though a new job of task τ_B is released at time 7 (Figure 7), it can only start within the next frame of super-TEE (C^{peak}) at time 9. We now discuss a requirement with regards to the delay between the release time and the start time for a task that any super-TEE must follow. Let us consider the task set in Table 5 where an execution instant of the modified task set (Table 6) after super-TEE construction is illustrated in Figure 8. At time 6, if a higher priority job preempts the super-TEE job for longer than 0.3 time units, task τ_B within C^{peak} frame will miss its deadline even though the super-TEE itself will not. To guarantee the timeliness of individual tasks τ_A and τ_B that forms a super-TEE, we must have at least one job of τ_A completely contained within each period of τ_B. We start with the following lemma.

LEMMA 1. *Let us consider a super-TEE candidate, which consists of the task-tuple* $\{\tau_i, \tau_j\}$. *If* $T_i \leq T_j$ *and* $2 \cdot T_i - gcd(T_i, T_j) \leq T_j$ *where* $gcd()$ *denotes the greatest common divisor, then each job of the super-TEE candidate with period* T_i *is guaranteed to meet the individual deadlines of tasks* τ_i *and* τ_j.

PROOF. We leverage cyclic scheduling of synchronous periodic tasks [22], which are invoked in frames. Each frame of length m that invokes a job of a task τ_k must start after the job arrival and end before the job's period to guarantee schedulability, also given by,

$$m + (m - gcd(m, T_k)) \leq T_k, \tag{6}$$

where T_k is the period of task τ_k and, gcd is the *greatest common divisor*. For our super-TEE candidate $\{\tau_i, \tau_j\}$, the period is set at $\min(T_i, T_j)$ (Section 4.3). To guarantee the timeliness of individual tasks τ_i and τ_j that form a super-TEE, we must have at least one task of τ_i (i.e., equivalent to a *frame*) completely contained within each period of execution of τ_j. Therefore, Equation 6 can be rewritten as $2 \cdot T_i - gcd(T_i, T_j) \leq T_j$, and the lemma holds. □

LEMMA 2. *Let us consider a super-TEE candidate, which consists of the task-tuple* $\{\tau_i, \tau_j\}$. *Then,* $C_{ij} < C_i + C_j$.

PROOF. Let us assume that the upper bound on the worst-case computation time by fusing the task-tuple $\{\tau_i, \tau_j\}$ into a super-TEE τ_{ij} is denoted by C_{ij}. From Equation 2, we have,

$$C_{ij} = C_{ij}^{peak} = \sum_{q=1}^{2} C_i^q + v_i + \sum_{q=1}^{2} C_j^q + v_j - (v_j^s + v_j^d),$$

$C_i = \sum_{q=1}^{2} C_i^q + v_i$, and $C_j = \sum_{q=1}^{2} C_j^q + v_j$. We define $C_j' = \sum_{q=1}^{2} C_j^q + v_j - (v_j^s + v_j^d)$. Since $C_j' < C_j$, we have $C_{ij} < C_i + C_j$, and the lemma holds. □

We are now ready to present a sufficient condition for schedulability of a super-TEE candidate for a given task set.

THEOREM 1. *Let us consider a task set* Ψ *that is deemed schedulable according to the LL limit (Equation 1). Further, let us consider a super-TEE candidate, which consists of the task-tuple* $\{\tau_i, \tau_j\}$, *where* $\tau_i, \tau_j \in \Psi$. *Let us refer to this task tuple as* τ_{ij}, *which can be modeled as discussed in Section 4.3. If lemma 1 holds, and* $C_i \geq C_j$, $T_i \leq T_j$, *and* $\frac{C_j}{C_j'} \geq \frac{T_j}{T_i}$, *where* $C_j' = C_j - (v_j^s + v_j^d)$, *then* $\Psi' = (\Psi \cup \tau_{ij}) - \tau_i - \tau_j$ *is also schedulable per Equation 1.*

Table 7: Original Task Set

Task (τ)	C^1	v	C^2	C	T
τ_1	0.2	0.7	0.1	1	2
τ_2	0.6	0	0	0.6	8
τ_3	0.1	0.8	0.1	1	2

Table 8: Candidate super-TEE profiles

Task-tuple	\$ Footprint	# Concurrent jobs	Execution Overlap	Rank
$\{\tau_1, \tau_2\}$	20%	1	0.6	2
$\{\tau_1, \tau_3\}$	10%	4	1	1
$\{\tau_2, \tau_3\}$	20%	1	0.6	2

PROOF. According to Lemma 2, we have $C_{ij} < C_i + C_j$. Given that Ψ is RM schedulable according to Equation 1, Ψ' is also schedulable if $U(\tau_{ij}) \leq U(\tau_i) + U(\tau_j)$. We know that

$$U(\tau_{ij}) = \frac{C_i + C_j'}{T_i}$$

Given that $\frac{C_j}{C_j'} \geq \frac{T_j}{T_i}$, then we can replace C_j' in the above equation with $\frac{C_j \cdot T_i}{T_j}$, and get,

$$\begin{aligned} U(\tau_{ij}) &\leq \frac{C_i + \frac{C_j \cdot T_i}{T_j}}{T_i} \\ &\leq \frac{C_i \cdot T_j + C_j \cdot T_i}{T_j \cdot T_i} \\ &\leq \frac{C_i}{T_i} + \frac{C_j}{T_j} \\ &\leq U(\tau_i) + U(\tau_j). \end{aligned}$$

□

An important consequence of Theorem 1 is that if the original task set is schedulable, and we only construct a super-TEE when Theorem 1 holds, the modified task set is also guaranteed to be schedulable. While we cannot at this point provide any guarantee on the schedulability of the modified task set if the original task set is not schedulable, our approach can sometimes make the task set schedulable as shown in the example below. We will also show that our approach can in fact improve schedulability under many scenarios as discussed in Section 7.

4.4.1 Example: Let us consider the task set in Table 7, which fails the RM schedulability test according to Equation 1. Applying our approach, the modified task set is shown in Table 9 and contains a task tuple $\{\tau_1, \tau_3\}$ fused to form a super-TEE according to Section 4.5, and an individual task τ_2. Although the super-TEE has a modified worst-case execution time and period, it passes the feasibility test outlined above. Since the utilization of this modified task set falls within the LL limit (Equation 1), the tasks are guaranteed to be schedulable using RM.

4.5 Finding Super-TEE Candidates

Now that we have explained how to construct a super-TEE and find its aggregated utilization, we turn our attention to selecting the

Table 9: Optimized Task Set

Task (τ)	C^{peak}	C^{nml}	T	l
$\{\tau_1, \tau_3\}$	1.2	1	2	1
τ_2	0.6	0.6	8	–

actual tasks best suited to form a super-TEE. The first step is to ensure that constructing a super-TEE does not negatively impact the schedulability of the entire system, as discussed in the previous section. Therefore, we apply Theorem 1 to all possible combination of task-tuples $\{\tau_i,\tau_j\}$ to check for a feasible taskset. Once the feasibility criterion is met, every possible combination of task-tuple $\{\tau_i,\tau_j\}$, $\tau_i, \tau_j \in \Psi$, is profiled for its execution pattern, concurrent job instances, maximum execution overlap duration, and combined secure cache footprint. This is a computationally intensive process, but only needs to be performed once offline. Given all the combination of task-tuples, our goal is to eliminate infeasible task-tuples and rank the remaining ones. Task-tuples whose secure cache footprint exceeds the maximum cache size of a core is removed from further consideration; architecturally, the execution overhead of an application significantly reduces if it experiences a low cache miss rate. By only considering task-tuples that are within the cache limit, we avoid unnecessary cache misses, thereby reducing memory fetches during application runtime and improve temporal predictability.

Since there may be a large number of task-tuples that are within the cache limit, we propose to prioritize the task-tuples by their concurrent job instances. For example, from Figure 3(b), we can graphically deduce that a task-tuple $\{\tau_1, \tau_3\}$ has a total of 4 concurrent job instances within the hyperperiod, while task-tuple $\{\tau_1, \tau_2\}$ has 1. Since the system saves CPU cycles every time the super-TEE of a task-tuple executes (Lemma 2), the higher the frequency of overlapping execution sections, the larger the increase in CPU cycle savings. For instance, in Table 8, since $\{\tau_1, \tau_3\}$ has max{# *Concurrent jobs*}, it receives rank 1 (highest priority). Ties are broken in favor of task-tuples with the smallest interval of overlapping execution sections, as smaller overlaps between execution intervals of two tasks indicate higher compatibility for sequential execution of the tasks under consideration [18]. Table 8 shows the ranked list of all task-tuples from our example task set (Table 7).

The steps to determine the tasks to fuse together are shown in Algorithm 1. The *TupleList* is the set of all task-tuples. Task-tuples whose cache footprint exceeds *FootprintLimit* are discarded. For each of the remaining task-tuples, `SORT_CONCURRENT_COUNT()` calculates the number of concurrent jobs if the tasks in the task-tuple were run in isolation till their hyperperiod, and sorts the tuples in a non-increasing order. Finally, ties are broken by comparing the size of overlaps to create a ranked *TupleList*.

5 CT-RM SCHEDULING ALGORITHM

To the best of our knowledge, there are no existing approaches to schedule super-TEEs in a hard real-time system while providing deadline guarantees. As such, we leverage an existing fixed-priority rate-monotonic scheduling policy (RM-FF) to schedule super-TEEs, along with other real-time tasks, on multicore systems. We opt for partitioned scheduling since it has the advantage of reducing the multiprocessor scheduling problem to scheduling on individual processors. In addition, since our task set consists of normal real-time tasks and super-TEEs, we modify RM-FF by changing the task

Algorithm 1 Task Fusion

```
 1: function TUPLE_OPTIMIZATION($tupleList$)                    ▹ Rank task tuples
 2:     for $\{\tau_i, \tau_j\} \in tupleList$ do
 3:         if CACHE_FOOTPRINT($\{\tau_i, \tau_j\}$) $> FootprintLimit$ then
 4:             REMOVE_TUPLE($tupleList, \tau_i, \tau_j$)
 5:     SORT_CONCURRENT_COUNT($tupleList$)                         ▹ Non-increasing
 6:     for $tuple1, tuple2 \in tupleList$ do
 7:         if $tuple1.overlap == tuple2.overlap$ then
 8:             SORT_OVERLAP_SIZE($tuple1, tuple2$)                ▹ Non-decreasing
 9: function FIND_CANDIDATE($\tau_i, \tau_j$)               ▹ Check feasibility: Theorem 1
10:     if $\frac{C_j}{C'_j} \geq \frac{T_j}{T_i}$ and $2 \cdot T_i - gcd(T_i, T_j) \leq T_j$ then return 1
11:     else return 0
12: ▹ Generate multi-frame task parameters for feasible super-TEEs
13: function CONSTRUCT_SUPERTEE($tupleList$)
14:     for $\{\tau_i, \tau_j\} \in tupleList$ do
15:         if FIND_CANDIDATE($\{\tau_i, \tau_j\}$) then
16:             Assign $C_{ij}^{peak}$ using Equation 2
17:             Assign $C_{ij}^{nml}$ using Equation 3
18:             Assign $l_{ij}$ using Equation 4
19:         else
20:             REMOVE_TUPLE($tupleList, \tau_i, \tau_j$)
21: function MAIN($tupleList$)
22:     CONSTRUCT_SUPERTEE($tupleList$)
23:     TUPLE_OPTIMIZATION($tupleList$)
24:     return
```

partitioning policy. The advantages of our modifications are apparent when discussed in Section 6.

We first define a *TupleList*, which consists of a list of task-tuples, e.g., (τ_i, τ_j), where τ_i and τ_j form a super-TEE (Section 4.5). We sort task-tuples in a non-decreasing order of periods to obtain the worst-case critical instant [23]. We divide the task-to-core assignment step into two phases (Algorithm 2). In the first phase, we leverage the existing RM First-Fit (RM-FF) partitioning policy, where tasks are assigned to a core until it is no longer RM schedulable, after which, the next core is considered [23], to assign tuples (super-TEEs) to cores. We set per-core admissible utilization bound to the LL limit (Equation 1). (A worst-case response time analysis can be used instead and is left for future work.) Lines [3-6] in Algorithm 2 performs the first phase of task-to-core mapping using the first-fit policy. In the second phase, we turn our attention to *RemTaskList*, which consists of tasks which are not part of *TupleList*. Lines [10-13] in Algorithm 2 performs the second phase of task-to-core mapping, again, using the first-fit policy after having sorted the tasks in a non-decreasing order of periods.

Algorithm 2 Task-to-Core Mapping

```
 1: function TASK_TO_CORE($TupleList, TaskList, Cores$)
 2:     $U_{max} \leftarrow$ LL limit                          ▹ Check for LL bound: Equation 1
 3:     for $\{\tau_i, \tau_j\} \in TupleList$ do                     ▹ Assign tuple to core
 4:         $\{\tau_i, \tau_j\}.Core \leftarrow$ FIRST_FIT($\{\tau_i, \tau_j\}, U_{max}$, Cores)
 5:         if $\neg\{\tau_i, \tau_j\}.Core$ then                    ▹ If tuple assignment fails
 6:             REMOVE_TUPLE($TupleList, \tau_i, \tau_j$)
 7:     $RemTaskList \leftarrow TaskList - TupleList$
 8:     $RemTaskList \leftarrow$ SORT_INCREASING_PERIOD($RemTaskList$)
 9:     ▹ Assign remaining tasks to cores
10:     for $task \in RemTaskList$ do
11:         $task.Core \leftarrow$ FIRST_FIT($task, U_{max}$, Cores)
12:         if $\neg task.Core$ then                              ▹ If task assignment fails
13:             $exit()$
```

Table 10: Summary of experimental results of synthetic benchmark (Figure 6) running on Rpi 3B using CT-RM and RM-FF with respect to percentage of feasible task sets as a function of task set utilization.

Util (%)	RM-FF	CT-RM
	Feasible task set (%)	Feasible task set (%)
100	100	100
150	100	100
200	87	96
250	72	86
300	21	48

6 EXPERIMENTS

In this section, we evaluate the benefits of our proposed approach by scheduling real-time synthetic benchmarks on an actual hardware platform.

6.1 Experimental Setup

For our hardware testbed, we use the Raspberry Pi 3B (Rpi 3B), a small computer with quad-core ARM Cortex A53 processor, 1 GB LPDDR RAM, and numerous sensors. It can run Linux and other non-trusted operating systems. It also extends support for ARM TrustZone. We used the Linux `RT_PREEMPT` Kernel v4.6.3 to build our prototype CT-RM by modifying the Linux real-time scheduling class `SCHED_DEADLINE`. We run the modified real-time Linux OS (as non-trusted OS) along with OP-TEE OS [13] (as TEE) on the Rpi 3B that extends the support for ARM v8 embedded virtualization. We also designed a task set generator to create synthetic benchmark applications as shown in Figure 10. The input to said task set generator are `util_factor`, `tee_tasks` and `tee_exec`, where `util_factor` represents the task set utilization level, `tee_tasks` denotes the number of tasks with TEE requirements in each task set, and `tee_exec` is the maximum percentage of the worst-case execution time of each task that has to be assigned for trusted execution in a TEE environment.

For the experimental results presented in Table 10, the task sets were generated over a range of utilization levels (`util_factor` = 100%, 150%, 200%, . . ., 300%) and, for each utilization level, we generated 100 task sets, each of which has a random number of tasks. The period (T_i) and worst-case execution time (C_i) of each task are randomly generated so long as the overall utilization of the task set remains within the corresponding utilization level. The worst-case execution time of a task τ_i is upper bounded by $\sum_{q=1}^{2} C_i^q + v_i$, where C_i^q denotes each non-secure execution segments, and v_i denotes the trusted execution interval. Each task set consists of (`tee_tasks` =) 60% of the tasks with TEE requirements and 40% ordinary real-time tasks. The trusted execution duration (v_i = `tee_exec`) of each TEE task is set at 60% of the worst-case execution time. The secure cache size limit is set to a high 90% of the maximum cache size to remove the effect of hardware-specific cache limit. We use all the 4 cores of Rpi 3B to run our experiments. Each experiment is carried out for one hyperperiod, the least common multiple of the periods of all the tasks in a task set. The results in Table 10 show an improved usable utilization bound by 12%, on average by CT-RM over RM-FF across all utilization levels (100%, 150%,. . ., 300%).

6.1.1 Case Study: We explain the implementation details for scheduling CT-RM on tasks whose attributes are shown in Table 9, along with another real-time task τ_4 with attributes $(2, 1, 1, -)$ (see Section 4.3). The task set is scheduled on a 4-core platform. All the tasks realize context-switching between non-trusted and trusted environments through an `INTERRUPTABLE` sleep duration. Each task starts as a completely fair scheduler (`CFS`) task which eventually switches to an `RM` task, and is distinguished from the other by the scheduler through a bit combination realized through a tuple (`cpu`, `OPTEE_on`, `OPTEE_task`) and communicated through a custom `syscall` to the kernel. The `cpu` denotes the assigned core, `OPTEE_on` categorizes CT-RM as the scheduling policy for the task and the `OPTEE_task` bit distinguishes a Super-TEE task ($\{\tau_1, \tau_3\}$) from other real-time tasks (with or without TEE requirements). Note, that CT-RM prohibits any task migration. Hence, we set the scheduler flag `NR_CPUS_ALLOWED` to #1 in the Linux scheduler. This informs the scheduler that the current task will never be up for migration. We validate the task-to-core assignment, the desired real-time characteristics and application's flow of execution by tallying the kernel log time stamps. Experiment log reveals that all the tasks complete their execution by the deadlines and behave as expected.

7 SIMULATIONS

Since architecture-specific design of Rpi3B constrains the maximum available secure cache dedicated for TEE usage, our hardware testbed is limited by the number of TEEs that can run simultaneously. Therefore, we extend the validation of our proposed approach (CT-RM) and assess its real-time performance against RM-FF in a simulated environment on randomly generated task sets. Similar to our hardware experiments, we use the task set generator over a range of utilization levels (`util_factor` = 100%, 150%, 200%, 250%,. . ., 1000%). For each utilization level, we generated 100 task sets, each of which has a random number of tasks, of which tasks with TEE sections have `tee_exec` ranging between 30% – 90%. For the reported simulation result, each task set consists of (`tee_tasks` =) 60% tasks with TEE requirements. The secure cache size limit is set to a high 90% of the maximum cache size to remove the effect of hardware-specific cache limit. Each simulation run tests the feasibility of the generated task sets using the steps listed in Algorithm 2.

Figure 9 reports the average results comparing the performance of CT-RM against RM-FF in terms of number of feasible synchronous task sets as a function of utilization levels with scaling cores. Our results indicate that the region of improvement with our approach over partitioned RM-FF with scaling utilization levels over increasing core count widens, indicating a scalable solution. While we observe a comparable performance between RM-FF and CT-RM for a 2-core system (4% average improvement in task set feasibility across all utilization levels), the effectiveness of our solution is more pronounced in 8-core and 10-core system (23% and 34% average improvement in task set feasibility respectively across all utilization levels). Overall, the trend shows an improved feasibility of up to 38% and, 18% on average over partitioned RM-FF across all utilization levels.

Since this work targets hard real-time systems, we further test the applicability of our solution on tasks with harmonic periods, a subset of synchronous task sets. Harmonic task sets are widely used in industry applications, ranging from cyber-physical systems [24–26] to control systems [27], as they allow 100% utilization to be realized when using fixed-priority scheduling policy [28]. Comparing the performance of CT-RM in terms of number of feasible harmonic

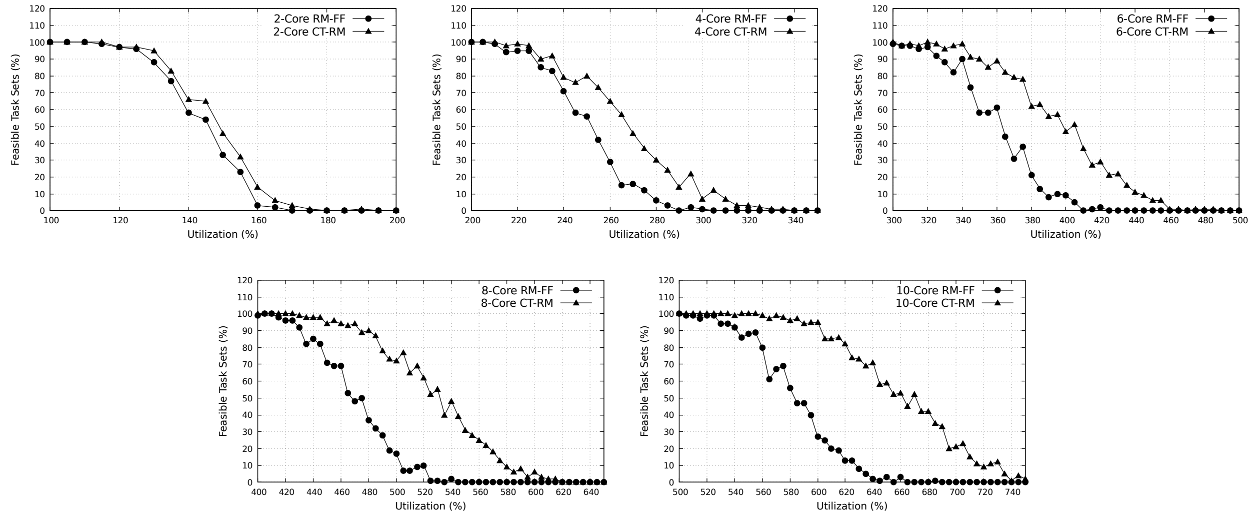


Figure 9: Simulation results showing the number of feasible synchronous task sets as a function of system utilization demand with scaling core count.

```
#define SCHED_DEADLINE 6 //Modified SCHED_DEADLINE for CT−RM
void *inc_x(void *x_void_ptr){ //Normal RM task body
...
attrX.sched_policy= SCHED_DEADLINE;
ret = sched_setattr(0, &attrX, flags); //set CT−RM params
...
while(++(*x_ptr) < 100); //normal computation
...
return NULL;}
void *inc_y(void *y_void_ptr){//super−TEE task body
funcA(); //Non−trusted code: Task τ1
funcX(); //Non−trusted code: Task τ2
usleep(...); //simulate self−suspension for data dependency
...
funcC()//Rest of non−trusted code: Task τ1
funcZ()//Rest of non−trusted code: Task τ2
}
void *inc_z(void *z_void_ptr){//fused TEE section
*'\colorbox{light-gray}{//Start TEE execution}'*
TEEC_InitializeContext(NULL, &ctx);
TEEC_OpenSession(&ctx,&sess,&uuid,..);
TEEC_InvokeCommand(funcB,...);//TEE call: Task τ1
TEEC_InvokeCommand(funcY,...);//TEE call: Task τ2
TEEC_CloseSession(&sess);
TEEC_FinalizeContext(&ctx);
*'{\colorbox{light-gray}{//End TEE execution}'*
...}
int main(){ //Main CFS task to spawn other SCHED_DEADLINE tasks
syscall(288, 1, 1, 0); //independent RT task to run on core #1
pthread_create(&inc_x_thread, NULL, inc_x, &x);
syscall(288, 2, 1, 0); //super−TEE task to run on core #2
pthread_create(&inc_y_thread, NULL, inc_y, &x);
while(...){}//wait for TEE execution
syscall(288, 2, 1, 1); //fused TEE section to run on core #2
pthread_create(&inc_z_thread, NULL, inc_z, &z);
// wait for tasks to end
return 0; }
```

Figure 10: Code snippet of our representative benchmark application modeling the super-TEE shown in Figure 6.

task sets as a function of utilization levels with scaling cores show improved feasibility of 20% on average over partitioned RM-FF across all utilization levels.

8 RELATED WORK

Security for embedded hardware and/or software is a main focus of recent work [29, 30]. For real-time systems, the emphasis is usually on the trade-off between real-time constraints and security levels [31, 32]. Hasan et al. [33] created a security policy to realize a fixed-priority sporadic server. Our work, however, is orthogonal to existing work and can be used in conjunction rather than instead of.

ARM TrustZone is a widely used platform-level security solution for many real-time embedded systems with security requirements. For instance, virtualization solutions [2, 3] allow Linux OS and TEE to run simultaneously while maintaining real-time performance. Pinto et al. [34] used the ARM TrustZone to run a low priority thread of a real-time OS over secure virtualization. OP-TEE [13], which is an open source port of TEE-specific design for Linux running on the ARM hardware platform, was used to run trusted sections alongside a real-time Linux environment [35]. Similarly, Pinto et al. [34] created a FreeRTOS based execution environment where the trusted code is run on the ARM TrustZone as a low priority thread of an RTOS. All existing work, however, ignore the overhead associated with TEE and its impacts on hard real-time deadlines.

9 CONCLUSIONS

We tackled the challenges associated with using TEE in hard real-time systems without affecting the security and isolation features of TEE, nor requiring source code modifications. We introduced the concept of super-TEEs, where multiple secure sections or application code that require TEE execution, are fused together to amortize TEE execution overhead while maintaining logical correctness and improving timing predictability through reduced I/O and switches between non-secure mode and TEE mode of execution. To schedule super-TEEs, we presented a sufficient condition for schedulability for uniprocessors and a fixed-priority task assignment and scheduling algorithm (CT-RM) for multicore systems. Experimental results on a real hardware platform show that CT-RM improves the usable utilization over RM-FF by up to 27% and, 12% on average, and are confirmed by simulations.

REFERENCES

[1] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 1, pp. 57–64, IEEE, 2015.

[2] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig, "Arm trustzone as a virtualization technique in embedded systems," in *Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya*, 2010.

[3] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the Key," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)* (M. Bertogna, ed.), vol. 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 4:1–4:22, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[4] D. C. Challener and D. R. Safford, "Encrypted file system using tcpa," Mar. 11 2008. US Patent 7,343,493.

[5] S. M. Darwish, S. K. Guirguis, and M. S. Zalat, "Stealthy code obfuscation technique for software security," in *Computer Engineering and Systems (ICCES), 2010 International Conference on*, pp. 93–99, IEEE, 2010.

[6] Y. Liu, K. An, and E. Tilevich, "Rt-trust: automated refactoring for trusted execution under real-time constraints," in *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 175–187, ACM, 2018.

[7] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "Trustzone explained: Architectural features and use cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pp. 445–451, IEEE, 2016.

[8] R. Pettersen, H. D. Johansen, and D. Johansen, "Secure edge computing with arm trustzone.," in *IoTBDS*, pp. 102–109, 2017.

[9] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 488–501, ACM, 2017.

[10] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, (New York, NY, USA), pp. 14–27, ACM, 2018.

[11] 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.

[12] ARM, "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[13] "OP-TEE (Open Portable Trusted Execution Environment)." https://www.op-tee.org/. Accessed: 2018-05-27.

[14] "GlobalPlatform Device Technology TEE Client API Specification." https://www.globalplatform.org/mediaguidetee.asp. Accessed: 2017-10-05.

[15] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[16] A. Díaz-Ramírez, P. Mejía-Alvarez, and L. E. Leyva-del Foyo, "Comprehensive comparison of schedulability tests for uniprocessor rate-monotonic scheduling," *Journal of applied research and technology*, vol. 11, no. 3, pp. 408–436, 2013.

[17] M. Nasri and G. Fohler, "An efficient method for assigning harmonic periods to hard real-time tasks with period ranges," in *2015 27th Euromicro Conference on Real-Time Systems*, pp. 149–159, IEEE, 2015.

[18] C. Roig, A. Ripoll, and F. Guirado, "A new task graph model for mapping message passing applications," *IEEE transactions on Parallel and Distributed Systems*, vol. 18, no. 12, pp. 1740–1753, 2007.

[19] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pp. 549–564, 2016.

[20] G. Irazoqui and X. Guo, "Cache side channel attack: Exploitability and countermeasures," *Black Hat Asia*, vol. 2017, 2017.

[21] V. Lesi, I. Jovanov, and M. Pajic, "Security-aware scheduling of embedded control tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 188, 2017.

[22] T. P. Baker and A. Shaw, "The cyclic executive model and ada," *Real-Time Systems*, vol. 1, no. 1, pp. 7–25, 1989.

[23] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations research*, vol. 26, no. 1, pp. 127–140, 1978.

[24] H. Li, J. Sweeney, K. Ramamritham, R. Grupen, and P. Shenoy, "Real-time support for mobile robotics," in *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, pp. 10–18, IEEE, 2003.

[25] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings, "Using harmonic task-sets to increase the schedulable utilization of cache-based preemptive real-time systems," in *Proceedings of 3rd International Workshop on Real-Time Computing Systems and Applications*, pp. 195–202, IEEE, 1996.

[26] T. Taira, N. Kamata, and N. Yamasaki, "Design and implementation of reconfigurable modular humanoid robot architecture," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3566–3571, IEEE, 2005.

[27] Y. Fu, N. Kottenstette, Y. Chen, C. Lu, X. D. Koutsoukos, and H. Wang, "Feedback thermal control for real-time systems," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 111–120, IEEE, 2010.

[28] C.-C. Han and H.-Y. Tyan, "A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms," in *Proceedings Real-Time Systems Symposium*, pp. 36–45, IEEE, 1997.

[29] K. Gai, L. Qiu, M. Chen, H. Zhao, and M. Qiu, "Sa-east: security-aware efficient data transmission for its in mobile heterogeneous cloud computing," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, p. 60, 2017.

[30] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *27th USENIX Security Symposium (USENIX Security 18)*, USENIX Association, 2018.

[31] Y. Ma, W. Jiang, N. Sang, and X. Zhang, "ARCSM: A distributed feedback control mechanism for security-critical real-time system," in *Proc. Int. Symp. Parallel and Distributed Processing with Applications*, pp. 379–386, July 2012.

[32] K. Jiang, A. Lifa, P. Eles, Z. Peng, and W. Jiang, "Energy-aware design of secure multi-mode real-time embedded systems with FPGA co-processors," in *Proc. Int. Conf. Real-Time Networks and Systems*, pp. 109–118, Oct. 2013.

[33] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni, "Exploring opportunistic execution for integrating security into legacy hard real-time systems," in *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pp. 123–134, IEEE, 2016.

[34] S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares, "Freetee: When real-time and security meet," in *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pp. 1–4, IEEE, 2015.

[35] R. Liu and M. Srivastava, "Protc: Protecting drone's peripherals through arm trustzone," in *Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*, pp. 1–6, ACM, 2017.