Parallelizing Trusted Execution Environments for Multicore Hard Real-Time Systems

Tanmaya Mishra

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Thidapat Chantem, Chair
Ryan M. Gerdes
JoAnn M. Paul

May 10, 2019
Arlington, Virginia

Keywords: Trusted Execution, Hard real-time systems, Scheduling

Parallelizing Trusted Execution Environments for Multicore Hard Real-Time Systems

Tanmaya Mishra

(ABSTRACT)

Real-Time systems are defined not only by their logical correctness but also timeliness. Modern real-time systems, such as those controlling industrial plants or the flight controller on UAVs, are no longer isolated. The same computing resources are shared with a variety of other systems and software. Further, these systems are increasingly being connected and made available over the internet with the rise of Internet of Things and the need for automation. Many real-time systems contain sensitive code and data, which not only need to be kept confidential but also need protection against unauthorized access and modification. With the cheap availability of hardware supported Trusted Execution Environments (TEE) in modern day microprocessors, securing sensitive information has become easier and more robust. However, when applied to real-time systems, the overheads of using TEEs make scheduling untenable. However, this issue can be mitigated by judiciously utilizing TEEs and capturing TEE operation peculiarities to create better scheduling policies. This thesis provides a new task model and scheduling approach, Split-TEE task model and a scheduling approach ST-EDF. It also presents simulation results for 2 previously proposed approaches to scheduling TEEs, T-EDF and CT-RM.

Parallelizing Trusted Execution Environments for Multicore Hard Real-Time Systems

Tanmaya Mishra

(GENERAL AUDIENCE ABSTRACT)

Real-Time systems are computing systems that not only maintain the traditional purpose of any computer, i.e, to be logically correct, but also timeliness, i.e, guaranteeing an output in a given amount of time. While, traditionally, real-time systems were isolated to reduce interference which could affect the timeliness, modern real-time systems are being increasingly connected to the internet. Many real-time systems, especially those used for critical applications like industrial control or military equipment, contain sensitive code or data that must not be divulged to a third party or open to modification. In such cases, it is necessary to use methods to safeguard this information, regardless of the extra processing time/resource consumption (overheads) that it may add to the system. Modern hardware support Trusted Execution Environments (TEEs), a cheap, easy and robust mechanism to secure arbitrary pieces of code and data. To effectively use TEEs in a real-time system, the scheduling policy which decides which task to run at a given time instant, must be made aware of TEEs and must be modified to take as much advantage of TEE execution while mitigating the effect of its overheads on the timeliness guarantees of the system. This thesis presents an approach to schedule TEE augmented code and simulation results of two previously proposed approaches.

Dedication

To my parents, brother, Ankita and Ambika. Thank you for everything.

Acknowledgments

I would first like to thank my advisor, Dr. Thidapat Chantem. Without her guidance, encouragement, constructive criticism and invaluable advice, none of this work would have been possible. I would also like to thank the other members of my advisory committee, Dr. Ryan Gerdes for providing insights into the security implications of the work presented here, and Dr. JoAnn Paul for comments and advice which have helped mold this work into its current final form.

No research is done in a vacuum, especially this work. I would like to first thank Anway, my research partner. His ideas and work provided the inspiration for the research presented here. I would also like to thank Pratham, Mahsa and Gaurang for their help and long discussions over many cups of tea for the last 2 years. My friends from my undergraduate days, including Gaurang, Omkar, Prathamesh, Aniket, among many others have constantly lent encouragement as and when I needed it. I am grateful to Madhura for her constant support, encouragement and for being a source of happiness through out the years.

Finally, but never the least I would like to thank my parents, my brother and my cousins, Ankita and Ambika who have kept me motivated, supported and helped me regardless of the situation. Without their love, hard work and sacrifices, none of this would have ever been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data is, undoubtedly, the most important commodity today. With the advent of the internet and the subsequent explosion in the number of interconnected devices that grow by the billions every year, data is being generated at an unprecedented rate. This could range from an individual's personal data to a large company's intellectual property or even a government's internal secrets. As such, it is obvious that a significant portion of all this information needs to be secured against both unauthorized access as well as tampering. After all, it would be catastrophic for something like national security related data to fall into hostile hands.

Real-time systems primarily focus on timing guarantees and correctness. When real-time design principles are applied to severely resource constrained systems, such as small low-powered embedded systems, security measures are usually added ad-hoc. While real-time system security has been looked into since quite some time, such as the work by Ahmed and Vrbnksy [7] and George and Haritsa [25], they are mostly tuned towards the application domain. Incorporating changes into the scheduler and scheduling algorithm itself, to make the approach more generalized, only started in the late 2000s, such as that by Xie and Qin [49]. Scheduling approach changes are necessary because security measures such as code obfuscation and encryption can incur large overheads that could make a schedulable system, unschedulable. Due to this, hard real-time applications drop security enhancements. This is a severe oversight as such systems are not only sensitive to attacks aimed at creating timing

violations, but also contain information such as encryption keys, or IP such as proprietary software or algorithms. Integrity and confidentiality of such data are necessary for the correct functioning of the system as well for maintaining the technological upper-hand.

An important security problem, especially in today's interconnected world, is trust. For example, website certificate authentication over HTTPS [22] has now been made compulsory by popular web browsers such as Chromium by Google [6]. It is now expected by web browsers that a website a user visits provides a cryptographically signed certificate from a trusted authority to ascertain its authenticity to the user. This allows the user to trust that the website they visit is the website they intended to visit. On the other hand, the website trusts the user after it authenticates themselves using a username/password combination. Similarly, in the case of streaming services such as Netflix or Hulu, DRM [46] protected content needs to be decrypted on the user's device. To ensure the user does not abuse this data, the service provider must utilize a variety of techniques to trust the user's device not to misuse their content.

One such technique is the use of Trusted Execution Environments (TEE) on the user device. TEEs are isolated from the OS with which the user interacts (rich OS such as Linux) and have strict hardware enforced access control policies. TEEs run their own OS which communicates with the rich OS in a controlled manner. In current generation SoC designs, TEEs are widely supported by hardware extensions provided by vendors such as Intel [26] and ARM [8]. Examples of critical systems that currently utilize TEE are payment methods such as Samsung Pay, which uses Samsung Knox [33], an ARM TrustZone based TEE, where tokens generated by the on-device TEE, authenticate virtual credit card transactions and are transmitted by a TEE controlled NFC (near-field communications) transmitter. The fact that financial institutions utilize this technology attests to its maturity and robustness. Further, for content streaming discussed above, Google's highly popular Widevine [5] highest

level of security (L1) is afforded through secure coprocessors or using TEE/ARM TrustZone based solutions.

ARM and Intel based SoCs power billions of devices such as smartphones and low-powered embedded systems and many support TEEs. This makes TEEs an attractive candidate for providing a well tested, standardized, hardware-supported ready-made solution for storing and computing sensitive data and algorithms present in many critical, hard real-time systems. For example, a UAV such as a quadcopter can contain proprietary flight algorithms, image processing algorithms and sensitive data such mission co-ordinates. If this data is accessible to the rich OS and should an attacker gain access to the rich OS, the loss of confidentiality of such critical data could be very harmful, especially in military applications. These, however, can be hidden inside the TEE and the rich OS can be provided limited access.

However, incorporating TEEs into existing real-time systems is not straightforward. TEEs, though supported by hardware, still create significant overhead (Section 4). Secondly, current TEEs are limited in both size and capabilities and entire applications cannot be directly run inside the TEE. If TEEs are to be utilized by real-time systems, there is a need to create algorithms that are aware of the peculiarities and overheads of TEE, reducing them as possible.

As such this thesis does not delve into the security advantages and pitfalls of TEE, which have already been widely studied and discussed in academia and industry. This thesis provides insight into how utilizing TEE can affect timing guarantees of hard real-time systems and present an approach to mitigate the problem [28, 32, 35].

This thesis is divided into the following sections:

1. Background information regarding hardware support for TEE, specifically ARM TrustZone and OP-TEE, an open-source TEE implementation utilizing ARM TrustZone.

2. Challenges that arise when using ARM TrustZone based TEE in a hard real-time system

3. A new real-time task model that aims to exploit the TEE setup time to improve overall task response time.

4. A new hard real-time global scheduling algorithm, ST-EDF, based on the task model which improves schedulability. Simulation results show an average improvement of 5-12% in deadline-misses over current-generation scheduling algorithms.

5. Simulation results of two state-of-the-art approaches that aim to improve the schedulability of TEEs and how they perform with respect to current generation general-purpose TEE unaware real-time scheduling algorithms.

6. A case study based on the PX4 autopilot flight stack.

Chapter 2

Background Information

Our novel task model and algorithm presented in Section 5 and the algorithms whose simulation results are provided in the Section 9, are inspired from the execution flow utilized by OP-TEE running on ARM TrustZone hardware extensions. While the algorithms are general and can be utilized for other TEE environments, OP-TEE being a standardized open-source solution, is the representative case study for this work's relevance.

2.1 ARM TrustZone

ARM TrustZone [8] provides a hardware virtualization platform that arbitrates access to hardware resources. Figure 2.1 provides a high level overview of ARM TrustZone. Its purpose is to provide isolation between the user facing rich OS and trusted OS running in an isolated environment, the TEE. Regardless of the privilege level of the rich OS, it is not provided access to TEE delegated resources which may include register banks, specific portions of cache, main memory and peripherals. The reference implementation of the firmware which controls the ARM TrustZone is the ARM Trusted Firmware which implements Secure Boot that sets the privilege level of the rich OS and the trusted OS during boot-up while cryptographically verifying the boot images.

To support the TEE environment, the TrustZone first extends the processor to have a secure mode. The secure mode is configured via the Secure Configuration Register (SCR) in the

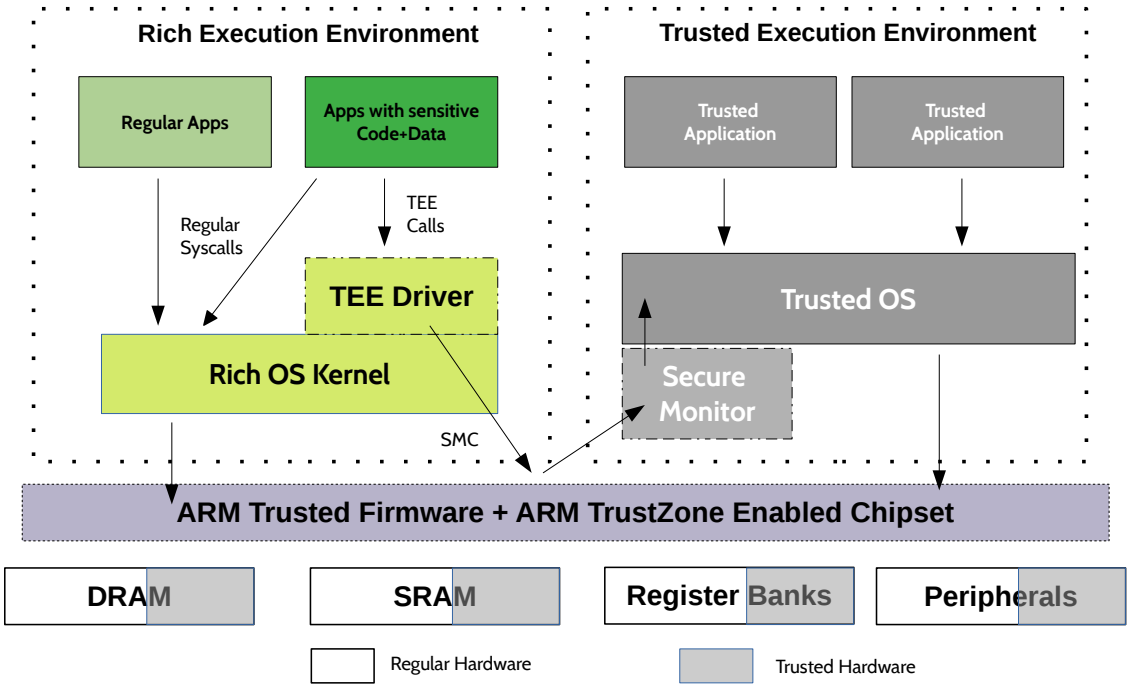


Figure 2.1: ARM TrustZone Architecture

coprocessor cp15. This mode is made unavailable to the rich OS by the firmware. The trusted OS runs when the ARM processor switches to its secure mode. This is handled by the secure monitor code section of the ARM Trusted Firmware. When a user process in the rich OS wishes that the Trusted OS perform run some code (bundled into a trusted application - TA), the userspace process issues a supervisor call (SVC) which is then handled by the rich OS kernel. A TEE driver is implemented in the kernel which then issues a secure monitor call (SMC). If a userspace process issues an SMC directly, it causes an exception since it can be only be issued by privileged code. The SMC is then handled by the secure monitor, which switches the processor into secure mode by resetting the NS bit in the SCR. Before doing so, it saves the state of the register banks. Once it enters the secure mode, it hands of execution to the trusted OS code which then runs the required trusted application. Returning from the secure mode to the non-secure mode is handled again by the secure

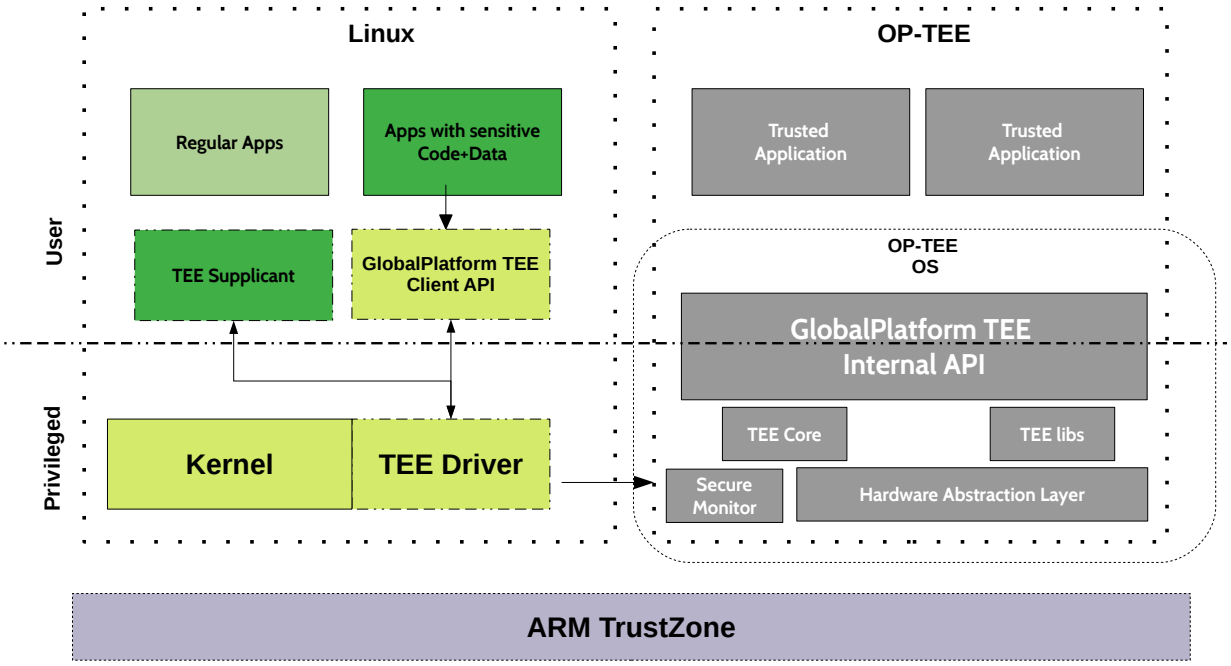


Figure 2.2: OP-TEE architecture specifics based on ARM TrustZone

monitor largely in the same manner but in the reverse direction.

The TrustZone ensures memory isolation by extending the memory management unit (MMU) such that it is aware of the processor mode and arbitrates the access to the secure memory. As such, it leaves the secure memory completely opaque to the non-secure rich OS, which, due to its privilege level is simply unable to ask the MMU for access to the secure memory. For handling interrupts, the interrupt controller first reads the SCR to ascertain the processor mode before deciding which kernel (secure or non-secure) must handle the IRQ or FIQ.

Peripherals and their access can also be handled such that the rich OS is totally blind to their operation and existence. This can be done by memory mapping these devices, during the Secure Boot sequence, to the secure memory region, which itself is handled by the MMU.

2.2 Open Portable Trusted Execution Environment (OP-TEE)

While the previous section shows the hardware modifications that ARM TrustZone provides to support TEEs, OP-TEE provides an implementation for TEE. An overview of its architecture is shown in Figure 2.2. OP-TEE consists of a secure kernel which handles the requests from the non-secure OS and runs Trusted Applications on demand, and a range of libraries for the TA to use including encryption and decryption algorithm implementations. OP-TEE is open-source and follows the standardized GlobalPlatform TEE specifications [1].

This specification requires an API to be used by a userspace process running in the rich OS (also called the client application) and a corresponding TEE API made available to the TA. The client API is supported by a TEE driver written for the Linux kernel. The client API provides features such as setting up shared memory buffers that are then copied to/form the trusted OS, opening a session to the required Trusted Application and invoking individual functions from the Trusted Application. The TEE API allows the TA to copy in/out the shared memory buffers and utilize OP-TEE OS internal libraries.

When a client application requests a TA to execute via the client API, the TEE driver sends out an SMC to the secure monitor. The secure monitor allows then suspends the client application and the rich OS, saves the state of the register banks, and switches the processor to secure mode. It then executes the OP-TEE OS kernel thread while loading the required TA. If the TA is stored in secure storage (part of the TEE and most secure storage location), the the OP-TEE OS handles loading of the TA. In the general case scenario, where the TEE is located in a special directory of the rich OS filesystem, a separate userspace process, the tee-supplicant aids in the process of locating and loading the TA into memory. The TA is signed along with the OP-TEE OS build, making it impossible for an attacker to run their code masquerading as a legitimate TA. The OP-TEE OS does not incorporate a scheduler

and relies completely on the rich OS scheduler to decide when it should run. Once the OP-TEE kernel thread is loads the TA, it jumps to the function being invoked by the client application. Once the function completes, the reverse process occurs to return execution back to the userspace process in the rich OS.

OP-TEE OS has provisions to handle both secure and non-secure interrupts. In both cases the TA context is saved before the interrupt is handled, which may include a switch back to the the non-secure processor mode in case of non-secure interrupts. Since TAs are scheduled by the rich OS scheduler, there is no guarantee that a TA will always be scheduled on the same core where it first ran after OP-TEE OS resumes its execution. The OP-TEE OS also has no provisions to preempt the rich OS processes.

Chapter 3

Related Work

As stated before, there is literature that considers real-time systems and security. Work such as that by Huang and Chen [29] consider correlation between real-time scheduling constraints and levels of security. Work by Hasan et al. [27, 28] consider sporadic servers or similar approaches to add security related execution into the system. Work by Jiang et al. [31] adds the energy consumption to the above to factors and uses FPGA co-processors. While all these techniques consider external mechanisms to monitor and detect security vulnerabilities, this work deals with using TEEs to augment an existing codebase while maintaining real-time guarantees. As such, there are no security related extra tasks, just a section of legacy code/data made to run inside TEE as seen later in 4.

While virtualization based security measures do exist, such as [48], hypervisor based solutions which utilize the ARM TrustZone have been proposed in literature by Frenzel et al. [24], Pinto et al. [44] which have timing measurement tools for utilizing in critical applications and some with real-time focus [43] where it is scheduled as a single thread under an RTOS. Application specific TEEs have been proposed, such as for IoTs [42]. This work focuses on maintaining deadlines when utilizing TEEs, which have not been considered before. Further, we deal with only the scheduling aspect of using a TEE (in our case a standardized, well studied TEE - OP-TEE), and not re-engineer a TEE environment suitable for a use case or scenario. OP-TEE is also used in work such as that by Liu and Srivastava [39] but does not explicitly consider or work towards maintaining deadlines in the face of TEE overheads.

Chapter 4

Challenges

TEEs may have a variety of security benefits, and for the most of their intended application domain, these advantages far outweigh the overheads created by TEEs. Considering the case of OP-TEE, overhead data of client API calls are presented in Table 4.1 on a Raspberry Pi 3 running a Linux kernel with real-time patches [4]. These API calls are the bare minimum required by a client application to connect to OP-TEE and open a session which loads the required TA(TEEC_OpenContext and OpenSession), calling a function implemented in the TA (TEEC_InvokeCommand) and then ending the session and closing the context to the TEE. The opening of a session to the TA is especially large which creates the bulk (upto 90%) of the overhead.

Moreover, TEEs do not share the same amount of resources, memory or compute capability that is available for the rich OS. For the Raspberry Pi 3, the RAM size is just 16 MiB and shared memory size is just 4 Mib [41]. Most TEE OS do not support many of the facilities expected of modern general-purpose operating systems such as memory segmentation, sophisticated scheduling frameworks and even floating point support, depending on the underlying hardware. This severely reduces the amount of code and data that could be split out from a legacy real-time codebase, and bundled into a TA. Thus, there is a need for judiciously separating sensitive code and data into the TAs. The rest of this thesis considers the task structure as shown in Figure 4.1. Sections of original code are stripped out and made to run inside the TEE. The temporal flow of execution is not modified from the original code,

Table 4.1: TEE execution overhead

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

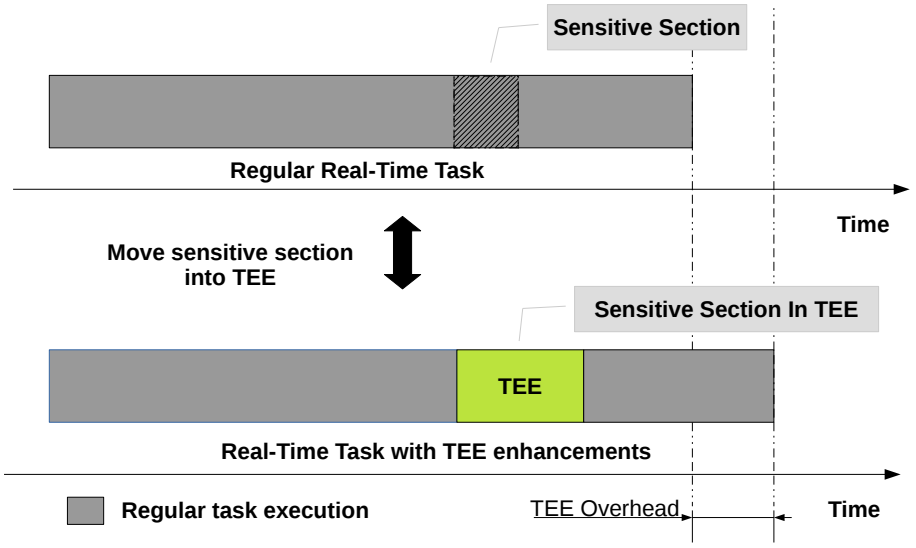


Figure 4.1: Task structures of a regular real-time task and the same enhanced with TEE.

just that the execution environment is changed.

It is obvious that the only way to reduce the runtime costs would be to put in a significant amount of work to optimize the implementation. These improvements would also need validation and standardization to ensure they do not adversely impact the security advantages afforded by the TEE. The entire process could take years to be deemed mature for real-world deployment. Further, some improvements simply cannot be made in software and are limited by the hardware available currently. For example, a large portion of the overhead caused by the TEEC_OpenSession is caused because of the need to load the TA executables from storage. This is bound by the speed and cost of storage mediums available today.

Thus, to utilize the significant security advantages of TEEs in current generation embedded platforms for the purpose of real-time systems to guarantee schedulability, real-time models and algorithms must take into account these overheads and work around them. Certainly, current generation real-time scheduling algorithms can work with TEEs without any modifications. However, the large overheads for TEEs must be judiciously scheduled to minimize their negative temporal impacts on the system.

Certain avenues for improvement could include:

1. As we are targeting multicore systems, we can use parallelism to our advantage and perform TEE and non-secure execution in tandem as much as possible. This is the new approach presented from Section 5. We provide system assumptions, a new task model (Split-TEE) and a new scheduling algorithm (ST-EDF) which takes advantage of this model to provide performance improvement. This approach requires minimal code changes so that the legacy code fits the new task model and can be readily scheduled by ST-EDF.

2. Utilize idle times created by task execution flow such as those caused by self-suspension and sleeping for inter-leaving TEE execution to reclaim CPU time. The T-EDF approach overview presented in Section 9.1 builds on this idea.

3. Reduce overall TEE usage costs by reducing the number of SMC calls. The CT-RM approach overview presented in Section 9.2

It should be noted that we present Split-TEE/ST-EDF as our novel contributions. The other approaches, at the time of writing this thesis, are being reviewed for publication and are joint efforts between the author and other researchers. We, thus, present only their simulation results and analysis as our contributions in this thesis.

Chapter 5

Split-TEE Task Model

This section presents a novel hard real-time task model that aims to reduce the overhead costs of using TEE augmented tasks such as that shown in Fig 4.1. This model is inspired from the OP-TEE API structure as presented in Table 4.1. We have an in-depth look at the task model as shown in the Figure 4.1, then provide a motivation to modify this model and create split tasks, followed by the Split-TEE task model.

5.1 System Assumptions

We consider a Symmetric Multiprocessing (SMP) system because the hardware testbed we studied and from which we obtained the OP-TEE overhead values(4.1), utilizes Linux and OP-TEE as its rich and trusted OS respectively. Since Linux supports SMP architecture since quite some time (Linux 2.0 [2]), this assumption is valid in a real-world scenario. We, thus, consider an SMP system with Uniform Memory Access (UMA) where there exists a private cache per core and a shared main memory where the time to access shared memory is the same for all cores, throughout this thesis. When migrating tasks from any core, the time to transfer to the shared memory is the same for all cores and the time to access this data is the same for all cores due to the SMP model. We, thus, can also assume that core-to-core unit data transfer time is the same regardless of which cores are communicating. For task migration costs, we limit our simulation analysis to the number of migrations necessary to

correctly schedule a feasible taskset. We do not consider the specifics of migration costs, such as different costs for different operations or situations. As such, our model does not allow task migration so this assumption does not affect the working of our algorithm, but when we compare with other algorithms, we simply note the difference in number of migrations. The actual cost calculation based on the per situation migration cost has not been considered.

5.2 Original Task Model

We first have a look at the task model (Figure 5.1) without any modifications. We use the same task structure as seen in Figure 4.1. Thus, the taskset consists of periodic tasks, denoted by τ_i that are represented by $(C_{i_1}^{nml}, \upsilon_i, C_{i_2}^{nml}, T_i)$ where $C_{i_1}^{nml}$ is the pre-TEE non-trusted execution interval, $C_{i_2}^{nml}$ is the post-TEE non-trusted execution interval, υ_i is the TEE execution interval and T_i is the period of the task. All execution intervals are worst case. Unless specified otherwise, the deadlines are equal to the periods. Certain considerations that exist here:

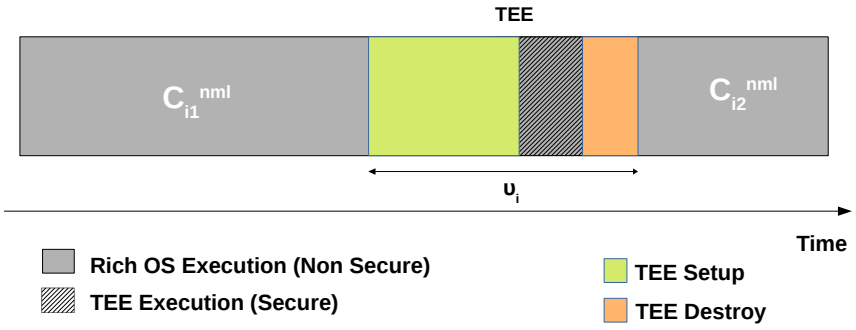


Figure 5.1: Task model without any self-suspension

- The tasks are considered to be independent of each other.
- The TEE section depends on the data generated by the $C_{i_1}^{nml}$ execution section.
- The second execution section $C_{i_2}^{nml}$ depends on the output generated by TEE.
- The TEE section consists of all the TEE specific code, including setup, execute and destroy of the TEE session.

We define a TEE task as one which has a non-zero υ_i and Normal task as one where $\upsilon_i = 0$

5.3 Motivation for split

Before we begin explaining how the the split-TEE task model is structured, consider a snippet of representative code where a TA is loaded and invoked from a regular task in Figure 5.2. This snippet is based on the OP-TEE TEE client API, but since the API itself is derived from the GlobalPlatform specifications for TEE, this case can be applied to all TEE implementations. This code runs in the rich OS to install a separate, newly built TA into the secure storage, and not the rich OS filesystem.

In this snippet, the 5 basic TEE client API commands are used. An interesting behavior to note is that the only TEEC_OpenSession actually takes in the UUID of the TA. The API actually loads the TA as part of the TEEC_OpenSession function by first asking the tee-supplicant to load from rich OS filesystem and then performing the SMC to begin the switch to secure mode. As noted earlier, this is one of the primary reasons for high execution time for the TEEC_OpenSession (Table 4.1). The actual execution of the TA is done much later when the TEEC_InvokeCommand API call is made. This is because the op operand variable must be populated before the TA can be called. This happens sometime after the

```
1  int install_ta_runner_cmd_parser()
2  {
3    TEEC_Result res = TEEC_ERROR_GENERIC;
4    uint32_t err_origin = 0;
5    TEEC_UUID uuid = PTA_SECSTOR_TA_MGMT_UUID;
6    TEEC_Context ctx = { };
7    TEEC_Session sess = { };
8    int i = 0;
9
10   res = TEEC_InitializeContext(NULL, &ctx);
11   if (res)
12     errx(1, "TEEC_InitializeContext: %#" PRIx32, res);
13
14   res = TEEC_OpenSession(&ctx, &sess, &uuid, TEEC
15           _LOGIN_PUBLIC, NULL, NULL, &err_origin);
16   ...
17   res = TEEC_InvokeCommand(sess, PTA_SECSTOR_
18           TA_MGMT_BOOTSTRAP, &op,&err_origin);
19   ...
20   TEEC_CloseSession(&sess);
21   TEEC_FinalizeContext(&ctx);
22   printf("Installing TAs done\n");
23   return 0;
24 }
```

Figure 5.2: Code snippet taken from install_ta.c, a part of OP-TEE's test suite

TEEC_OpenSession. This behaviour is important to note because, at least until after the TEEC_OpenSession call, there is actually no need to run this code inline as is seen in the snippet. Since the inline dependency need only be met for the actual invoking of a TA function, all TEE related code before this point can be made to run in parallel, while still maintaining logical flow of the program.

The TEE interacting code can, thus, be logically removed and moved into a separate thread of execution. There are three immediate advantages to this:

1. The thread containing TEE specific code can run in a different core.

2. The two threads can be started simultaneously. This allows the TEEC_OpenSession code to essentially warm up a core with the TA and wait until the data dependency for the TEEC_InvokeCommand is met and then immediately continue execution.

3. The core which is running the regular execution thread can simply execute some other waiting task while this thread waits for the output of the TEEC_InvokeCommand

Overall, based on the data from Table 4.1, there can be up to 95% reduction in the time for which the regular thread would have originally waited for the TEE API calls to complete. However, since the code is now running independently of each other and on different cores, without any form of bounding or synchronization, the regular thread could simply keep waiting for data because the TEE specific thread is stalled, possibly due to preemption, to a point where it misses its deadline. Thus, there the task model and the scheduling policy must carefully consider this situation, while maximizing the number of CPU cycles saved that is afforded by this model.

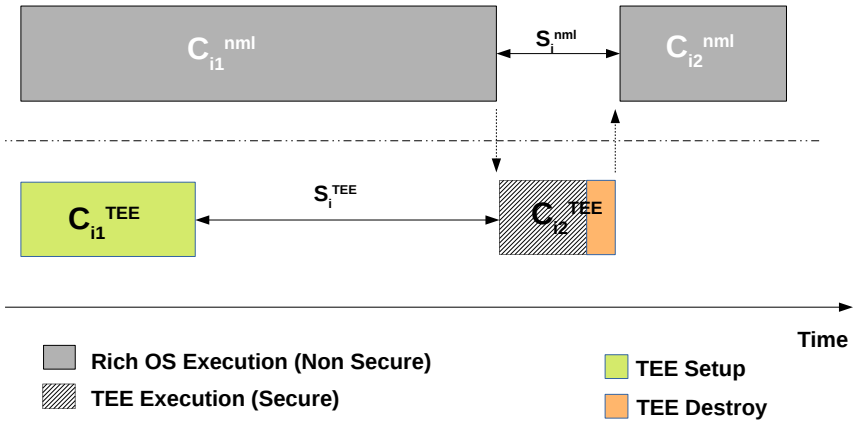


Figure 5.3: Split-TEE Task Model

5.4 Model after split

We now modify the original model into our split-TEE model as seen in Figure 5.3. Based on the above discussion, we first split υ_i into two parts. We define $C_{i_1}^{tee}$ as the time of TEE setup (context and session open) and $C_{i_2}^{tee}$ is the cumulative execution time for TA invocation and destroy. For sake of simplicity, we consider invocation and destroy together since destroying TEE context and session has very low overhead, so clubbing the two together can reduce the the number of scheduling decision by 1, with minimal impact to performance gain. Thus:

$$\upsilon_i = C_{i_1}^{tee} + C_{i_2}^{tee} \tag{5.1}$$

We now take every task τ_i of the original taskset and, according to Figure, we split it into two corresponding tasks (in case the original task had a non-zero TEE section):

1. Non-trusted execution task τ_i^n (normal task), which has a WCET of $C_i^{nml} = C_{i_1}^{nml} + C_{i_2}^{nml}$, wait time of S_i^{nml} where it waits for the data from the TEE task. In case the

original task had zero υ_i, $S_i^{nml} = 0$. The period is inherited from the original task and will remain as τ_i, which is also the deadline for this task. Hence it is formally the tuple,

$$\tau_i^n = (C_{i_1}^{nml}, S_i^{nml}, C_{i_2}^{nml}, T_i) \tag{5.2}$$

2. Trusted execution task τ_i^t (TEE task), which has an execution time of $C_{i_1}^{tee}$ followed by a wait time S_i^{tee} where it waits for the data from the normal task to be ready after $C_{i_1}^{nml}$. Finally, its followed by the the remaining execution time of $C_{i_2}^{tee}$. The period of this task will remain the same as that of the original, that is T_i, however, this task will have a relative deadline, $D_i^{tee} < T_i$. It is obvious that $C_{i_2}^{tee}$ must complete, in the worst case, such that there is still enough time remaining for the normal task to complete execution. Thus,

$$D_i^{tee} = T_i - C_{i_2}^{nml} \tag{5.3}$$

Formally, this task can be denoted by the tuple:

$$\tau_i^t = (C_{i_1}^{tee}, S_i^{tee}, C_{i_2}^{tee}, D_i^{tee}, T_i) \tag{5.4}$$

Thus, for an n task original system, where k tasks had TEE sections, the new taskset will now contain $n + k$ tasks. We use the term jobs for each task instance.

Chapter 6

ST-EDF: A Multicore Scheduling Algorithm for Split-TEE Tasks

The specifics of TEE operation, as discussed in Section 2, where the client task calls the TA and then yields the processor, is similar to the behavior of self-suspending tasks. There has been some work [15, 16, 29, 38] for both soft and hard real-time systems featuring tasks with self-suspensions. A task usually self-suspends when it interacts with external memory or devices [37]. However in the case of TEE, even in the simplest scenario of a unicore processor running only one task, when a task calls the TEE, the processor is not waiting for data to arrive from some external source. Instead, the processor itself switches into secure mode and continues executing the TA. Thus, the behavior of the system, internally, does not match the semantics of truly self-suspending system.

Instead, the split-TEE task model presented in Section 5 can be seen as a hard real-time task model which contains intercommunication section between pairs of tasks. There has been some work which considers task precedence [30, 34, 47] consider Directed Acyclic Graphs (DAG) based models and while [45] also considers the task intercommunication time. While these models provide a fine grained analysis with varying degrees of expressiveness, considering our model has a consistent task communication flow, these models are overly complex for our scheduling needs. Further, these models do not have explicit hard real-time scheduling guarantees. Instead, our approach simply bounds the execution and delays during communi-

cation to its worst case. This is done offline, explained later in this section, and then utilized when scheduling under our algorithm, ST-EDF.

In literature, several real-time scheduling policies [9, 10, 13, 18, 20, 21, 23] have been explored. Global-EDF (Global Earliest Deadline First) is a well-studied dynamic priority multicore scheduling algorithm [11]. G-EDF extends the EDF scheduling policy, which is known to be optimal [36] for single core hard real-time execution. G-EDF utilizes the same policy applied to a global queue populated with all ready jobs, and schedules the earliest deadline jobs on to the cores. G-EDF however has not been proven to be optimal for multicore systems. G-EDF, also, does not have any bounds on the number of times a job is migrated between the cores. The high cost of migration [14], combined with large number of preemption made possible since a single global queue is used which is populated with all released and ready jobs makes the usage of G-EDF less favorable. Instead, partitioned based mechanisms are much easier and have zero migration costs. Here tasks are partitioned offline using, while run against well-studied single core execution policies such as EDF or RM which are simple to implement and have very low overhead. However, all partitioned policies have much lower utilization bounds than G-EDF.

Instead, a middle ground approach would be most suitable for our task model. We base our approach on the restricted migration variant of EDF, r-EDF, proposed by Baruah et.al. [12], where pinning to core is done on a job-to-job basis but job migration is disallowed. This approach has utilization bounds equal to, or higher than, a fully partitioned policy. As we have no requirement that subsequent jobs of a task continue on the same core, we use r-EDF as the basis for our scheduling approach.

6.1 Wait time bounds

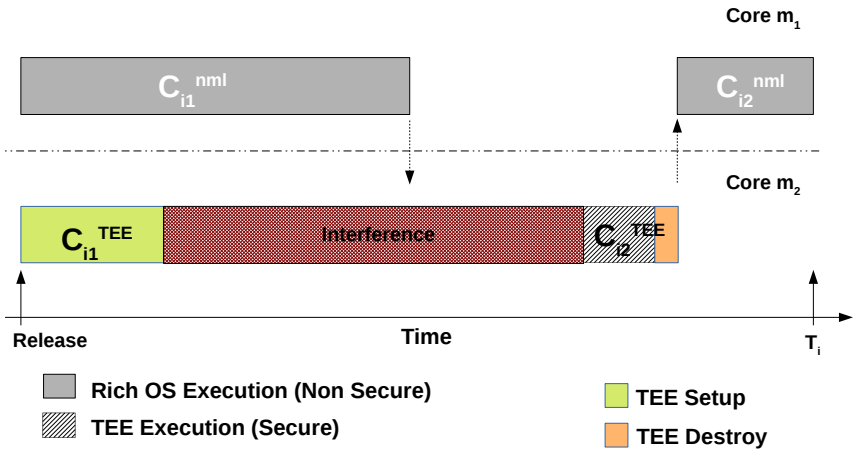


Figure 6.1: Worst case interference for TEE job

As seen in the split-TEE task model definition (Section 5.4), the runtime dependent factors that can affect the schedulability of the system, are the wait times for the normal and TEE tasks. Since the jobs of these tasks run independently from each other, on different cores which can have very different workloads which vary with time, it can lead to near infinite scenarios. While taking the worst case interference caused to the TEE job with no interference of the normal job, which subsequently causes the normal job to finish just by deadline, as seen in Figure 6.1 would be the default mechanism to model the wait times for our scheduling approach, it is unnecessary and makes the scheduling decision more complex.

Instead, we tackle the problem by looking at the bare minimum empty space on the cores required to schedule without a deadline miss. Our approach is to look at the time needed to be available on both cores, the one running the TEE task and the one running the normal task, such that the normal task meets its deadline. This is because the deadline assigned to the TEE task is an artificial one, to ensure that the normal task has enough time to end

execution by its deadline. We extend this concept to decide the wait times to be considered for the acceptance test in our scheduling algorithm.

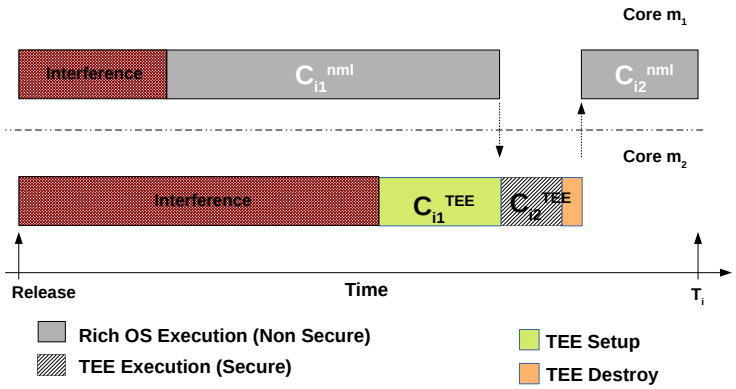


Figure 6.2: Worst case Interference for both jobs

Consider the execution scenario presented in Figure 6.2. This situation is created when there is worst case interference for both tasks in their respective cores. This situation represents the case where any further interference (such as preemption caused by higher priority jobs on the same core) would both tasks would miss their deadlines. Any interference on core m_1 would delay the data sent to the TEE job and cause it to miss its deadline, while causing an obvious deadline miss for the normal job. Similarly any interference $m2$ would cause the TEE job to delay sending data back to the normal job and cause both jobs to miss their deadline.

Hence, if the scheduling approach provides enough time for the execution to the normal and TEE jobs and provisions for the wait times such that:

$$S_i^{tee} = 0,\ S_i^{nml} = C_{i_2}^{nml} \tag{6.1}$$

The jobs will complete without a deadline miss.

6.2 ST-EDF

Our scheduling policy, ST-EDF is presented in Algorithm 1. Once jobs are pinned to a core, regular EDF is used for scheduling, where jobs yield when they need to communicate with their corresponding normal/TEE jobs and return to ready state when their data dependency is met. For pinning to cores we use the following two policies:

6.2.1 Core validity check

The FitCheck function in Algorithm 1 implements the validity check policy for ST-EDF. It uses the sufficient utilization based acceptance test used by most partitioning based EDF algorithms. However, our acceptance test must consider the case shown in Figure 6.2. Since the TEE job execution and job time is usually much shorter than the overall execution time of a normal job, in case the job-under-test (JUT) is a normal job, we consider the total $C_i^{nml} + S_i^{nml}$, where S_i is calculated using the result from Section 6.1 while calculating the utilization of the job.

In case the job is a TEE job, we consider the density of these jobs where the D_i^{tee} is provided in Section 5.4. As seen in Section 6.1, S_i^{tee} need not be considered.

6.2.2 Core pinning

The Assign function in Algoritm 1 implements our actual core pinning policy. While the previous policy checks whether a core is a valid target to pin a job, we must consider the state of all cores before we actually pin it to the core. We use the Best-Fit policy as our base policy, since Best-Fit is known to work better than other policies with widely varying task execution times. Considering our split task model has differently sized TEE and normal jobs,

this makes most sense for our scheduling needs. We, however, wish to maximize parallelism between corresponding TEE and normal jobs. So we try to these jobs to different cores first, before pinning them together on the same core. In such a case, the behavior reduces to r-EDF scheduling the original task model, i.e., the overall response time of the normal job does not change between our policy and r-EDF. The only difference would be that the TEE setup would take place before the normal job begins since the priority is decided by the deadlines.

Our sufficient condition for acceptance may be conservative, however, as is seen from the results, we still outperform r-EDF and G-EDF, even in overloaded conditions.

Algorithm 1 ST-EDF

```
1: JL ← set of newly released jobs
2: P ← set of m cores
3: function FitCheck((core, Job))                    ▷ Checks whether job will fit core
4:     if Jobis normal Job then
5:         util = (Job.C^nml + Job.C_2^tee)/Job.T
6:     else if Jobis TEE Job then
7:         util = (Job.C^tee)/Job.D
8:     if util ≤ core.remUtil then                   ▷ Remaining Utilization Space
9:         return Yes
10:    else
11:        return No
12:
13: function Assign(JL)                              ▷ Pin
14:    for J_i ∈ JL do
15:        if J_iis normal job then
16:            J_i^n ← J_i
17:            Find J_i^t ∈ JL                       ▷ Corresponding TEE job
18:        else                                      ▷ If J_i is a TEE job
19:            J_i^t ← J_i
20:            Find J_i^n ∈ JL                       ▷ Corresponding normal job
21:        Sort P in non-decreasing util order       ▷ Implement Best-Fit
22:        for p_j ∈ P do
23:            if J_i^n.assigned ≠ 1 then
24:                if FitCheck(p_j, J_i^n) then
25:                    p_j.ready ← J_i
26:                    J_i^n.core ← p_j              ▷ Core number saved
27:                    J_i^n.assigned ← 1
28:                    Update core util
29:            if J_i^t.assigned ≠ 1 & p_j ≠ J_i^n.core then   ▷ Check if not the same core
30:                if FitCheck(p_j, J_i^t) then
31:                    p_j.ready ← J_i^t
32:                    J_i^t.core ← p_j
33:                    J_i^t.OrigD = P_i − C_{i_2}
34:                    J_i^t.assigned ← 1
35:                    Update core util
36:        if J_i^t.assigned ≠ 1 then                ▷ If J_i^t can't be assigned to different core
37:            if FitCheck(J_i^n.core, J_i^t) then
38:                J_i^n.core.ready ← J_i^t
39:                Update core util
40:        Pop(JL, J_i^n, J_i^t)
41:
42: procedure Schedule(P)
43:    while true do
44:        if NewJobs then
45:            Assign(NewJobs)
46:        for p_j ∈ P do
47:            EDF(p_j)                              ▷ Run EDF on each core
```

Chapter 7

Simulation tool

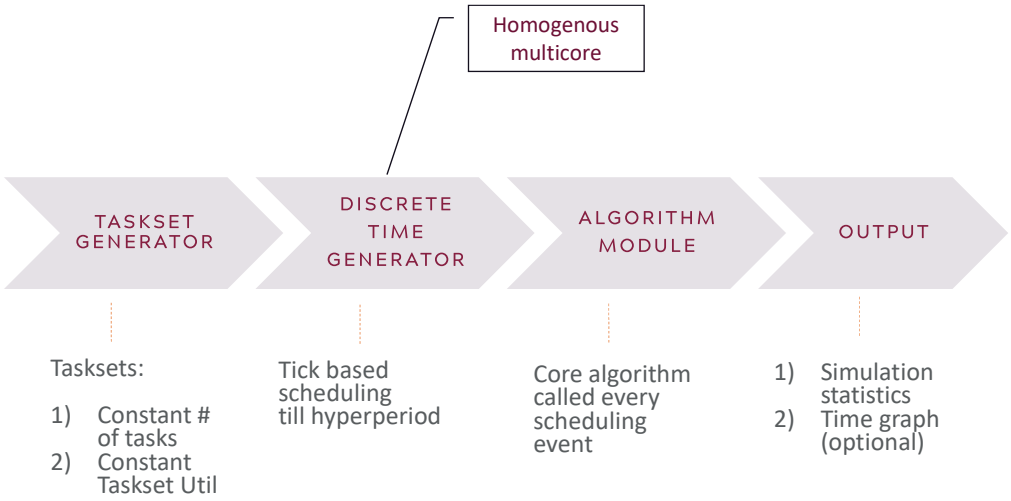


Figure 7.1: Overview of our custom multicore simulator tool

Before we present simulation results for our scheduling algorithm, this section provides a brief overview of the simulation tool we designed which is used for all the simulation results provided in this thesis.

Figure 7.1 shows the flow of the simulator. It consists of a task set generator that creates synthetic task sets which have randomized values, to remove any form of bias but adhering to certain parameters such as task set cumulative utilization, per task utilization bound and number of tasks. Consider the situation where the simulator is asked to generate a taskset with only a fixed total task set utilization, where utilization of any task is the ratio of the task execution time and its period. There are many forms of bias that could result

if anything but the total taskset utilizations is fixed. For example, if the number of tasks to be generated was fixed at a particular value and not kept random for different tasksets being generated, it could lead to situations where all tasks fit into cores (under a partitioning scheme such as First-Fit) if the number of tasks was large (smaller utilization chunks), or don't fit if the number of tasks were too small. This extra factor (number of tasks) would lead to much higher or much lower average failure rate for a scheduling algorithm leading to a skewed analysis. It thus must be ensured that such secondary patterns do not influence any batch of simulations which necessitates randomizing all parameters which are not specified as constant or fixed. However, it must be noted that the randomization is bounded based on the situation. For example, if the task to be generated has to have at most 40% utilization value, a random period would be decided for the task but the execution time would be bounded such that the task utilization would never exceed 40%. The task data generated includes the task execution time, the point where TEE execution begins and its length, the point where self-suspension begins and its worst case length, and the period and deadline of these tasks.

The bulk of the simulator consists of a framework which creates discrete timing ticks, considering a homogeneous multicore system where the number of cores are provided. It keeps track of currently running task instances and the cores on which they are running or supposed to run. The framework maintains the instance runtime parameters, such as how much execution time has elapsed. Based on the runtime parameters, such as if the task has self-suspended or is about to start TEE execution, the framework generates timing events.

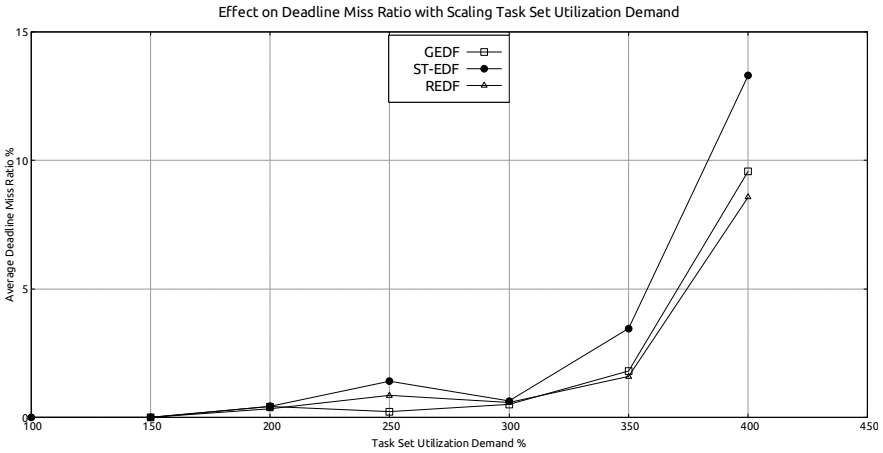
These timing events are captured by the algorithm module, which implements the scheduling algorithm logic and reshuffles the task queues and decides the next task to run. These algorithms can be global or run on a per-core basis.

The simulator outputs simulation statistics such as the number of deadlines missed and the

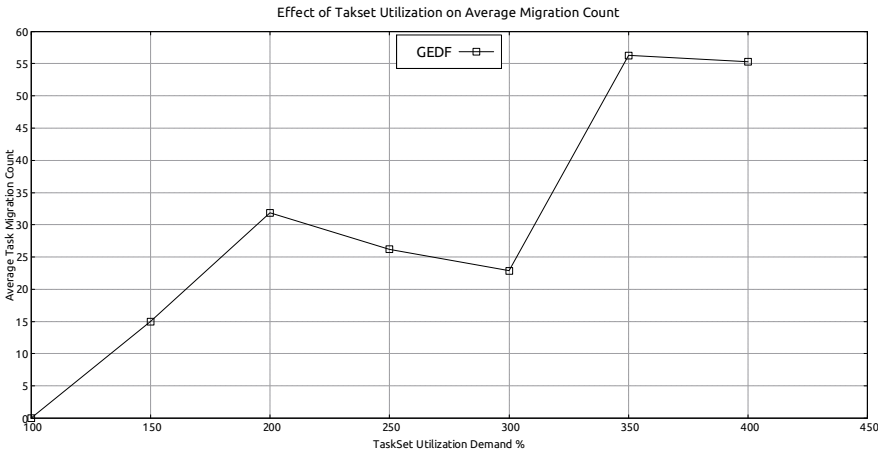
number of migrations necessary for correct scheduling. The simulator can also generate per-core timing graphs for visualization.

Chapter 8

ST-EDF simulations

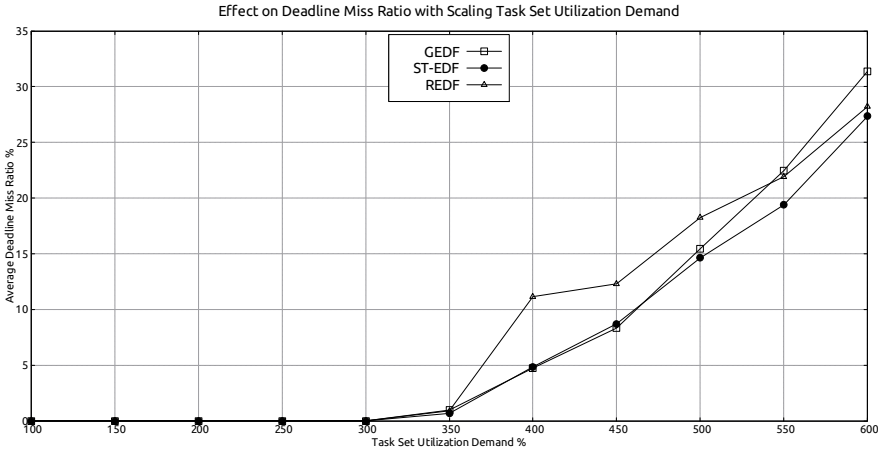


(a) Deadline miss ratio% for 5 tasks on 4 cores, 50% of execution length is TEE

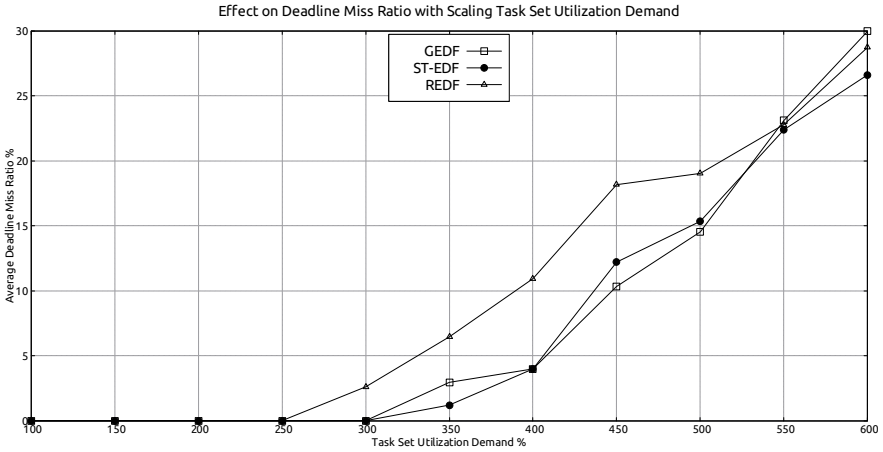


(b) Migration count for GEDF 5 tasks on for 4 cores, 50% of execution length is TEE

Figure 8.1: ST-EDF Simulation results



(d) Deadline miss ratio% for 10 tasks on 4 cores, 30% of execution length is TEE



(e) Deadline miss ratio% for 10 Tasks 4 cores, 50% of execution length is TEE

Figure 8.2: ST-EDF Simulation results cont.

We now present simulation results for ST-EDF in Figure 8.2. Since our scheme is based on pinning jobs to cores, we evaluate against the average deadline miss ratio with scaling utilization. Our scenario starts off as the same as that for T-EDF with 5 task running on 4 cores. We generated 100 such tasksets. Figures 8.1(a) and 8.1(b) are generated based on them where each taskset have 60% of the tasks having a TEE section and 50% of their execution time is spent inside the TEE. In such a scenario, it is seen that the 3 scheduling algorithms have nearly the same deadline miss ratios (Figure 8.1(a)) with scaling taskset utilization. In fact, there is a drop in performance with ST-EDF by about 1-3%, which is to be expected as ST-EDF artificially increases the number of tasks in the system by splitting, and under light loads there will be slight performance drop. However, as seen in Figure 8.1(b), G-EDF has upto large migration count while both R-EDF and ST-EDF do not suffer from this issue. We can conclude that ST-EDF does not perform the best under low load situations, but the performance drop is not severe.

We then scaled the number of tasks to be double, that is 10 tasks and ran under the same simulation conditions as before. We now notice an inversion in deadline miss ratio over the previous case and an appreciable increase in performance. For our base case of 30% of task execution time inside the TEE, we get upto 7% improvement in performance over both G-EDF and R-EDF in case of 50% of task execution time inside TEE, an improvement of up to 10%. This improvement is not as dramatic as T-EDF but it must be noted that ST-EDF is an improvement for the worst case situation of T-EDF and CT-RM, where they both devolve to running as R-EDF (their base case) since there is no suspension time and no task fusion.

It must be noted that we don't go past the 50% execution length inside TEE because of the nature of the ST-EDF algorithm. Since the TEE and normal split tasks are scheduled at the same time, going past 50% would result in the same situation as the inverted case, i.e,

from a scheduling perspective, the case of 70% non-trusted execution and 30% inside TEE execution and the case of 70% TEE execution and 30% inside TEE execution are the same.

Chapter 9

Two existing approaches to TEE scheduling - A simulation study

This section deals with 2 existing but orthogonal approaches to TEE scheduling. They can be used in conjunction with each other, however, they utilize different mechanisms for improving real-time tasksets which have have been augmented with TEE sections as seen in Figure 4.1. While the approaches themselves are not a contribution of this thesis, our simulation results presented provide insights into the applicability of these approaches.

9.1 T-EDF

9.1.1 Overview

Tasks running on a CPU core may pause execution for a variety of reasons. These could include intercommunication between tasks, synchronization requirements or preemption by other tasks. However, tasks can also self-suspend when they need some data or action to be completed, sometimes even waiting for an off-chip resource. For example, a task polling for sensor data spends most of its time self-suspended while it waits. This delay could be very small but could range up to orders of magnitude larger (μs - ms) than current generation processor cycle times (ns - μs). This is further exacerbated by the relatively slow peripheral

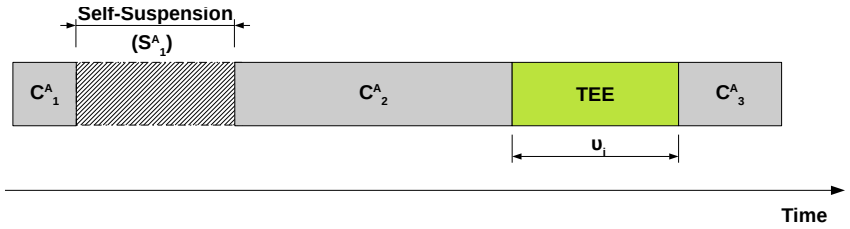


Figure 9.1: Task model for T-EDF

I/O communication mechanisms such as UART or SPI.

This issue for real-time systems is that self-suspension times are erratic and difficult to predict. Classically, for hard real-time systems, suspension times were bundled into the worst-case execution times (WCET) of the tasks themselves. While this fits well into existing real-time task models, this leads to overly pessimistic bounds for schedulability reducing the usability of the system. Recent work in self-suspension aware scheduling algorithms for multicore processors [15, 16, 29, 38] consider self-suspension separately from execution times, to give tighter schedulability analysis.

T-EDF was inspired from the PX4 [3], an open-source flight stack for UAVs, task model structure which consists of tasks self-suspending at the start while waiting for data from sensors or other tasks to be loaded into a shared message bus, and then continues execution, loading results back into the message bus for other tasks in the system to act upon. In such cases, the flight code may contain sensitive algorithms which are partitioned out into TAs and called inline from the original code. The task model, for T-EDF is presented in Figure 9.1.

For multicore systems, global scheduling algorithms such as global EDF (G-EDF) [11] have better utilization bounds than partitioned scheduling algorithms, however, suffer from large migration costs [14]. On the other hand, partitioned algorithms reduce runtime scheduling costs by running pre-partitioned tasks in different cores. To most efficiently utilize the self-

Table 9.1: Simulation use case scenarios

#	# TEE tasks	TEE length (υ_i)	Self-suspension length(S_i)
1	avg	short	short
2	high	long	long

suspension time of tasks, T-EDF is derived from a job based partitioning scheme, restricted-migration EDF (r-EDF) [12], which restricts the migration of jobs but allows different jobs of the same task to be pinned to different cores. T-EDF extends this by allowing TEE sections to migrate while pinning the non-secure portions of the code to the core.

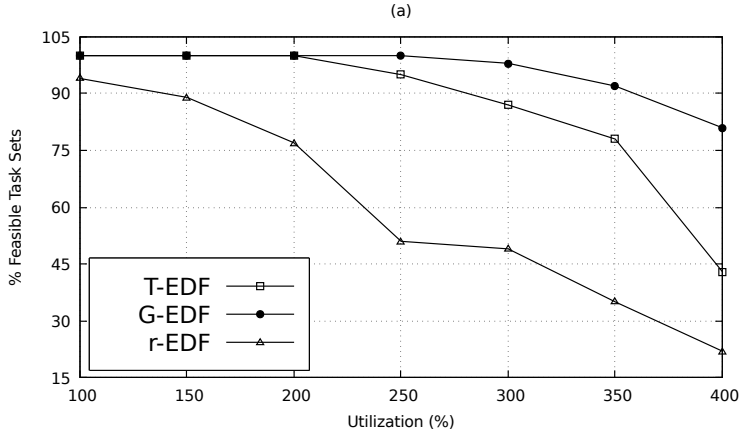
This allows T-EDF to opportunistically push TEE sections into the self-suspension times other tasks on other cores. To further improve the chances of the such a situation taking place, the algorithm proposes a novel metric DOPS (degree of parallel suspension intervals) which prefers that tasks with suspension intervals that will occur together (or close together) are pinned to the same core.

9.1.2 T-EDF simulation

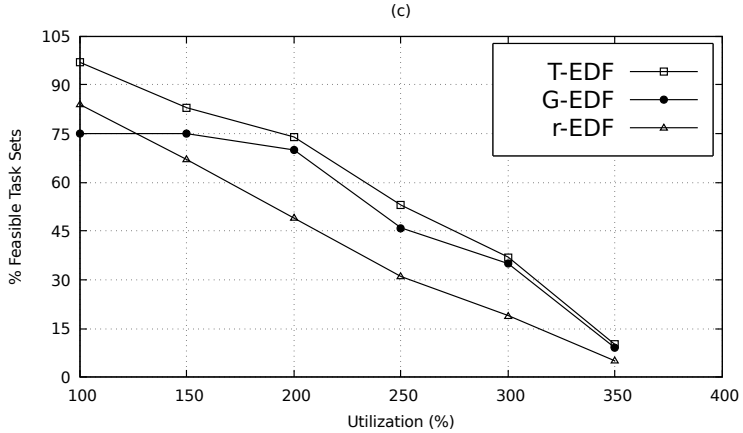
The following simulation results assess the real-time performance of T-EDF against G-EDF and r-EDF using randomly generated task sets. We compare the performance of T-EDF with both r-EDF and G-EDF as T-EDF is based on r-EDF. In addition, since T-EDF allows migration of TEE sections, we compare and contrast it against G-EDF.

Table 9.1 represents 2 different variants of task sets that are used to compare T-EDF against the existing G-EDF and r-EDF. We use these two variants since they show the opposite ends of the spread of the TEE utilization scenarios. While #1 scenario is for the case where the

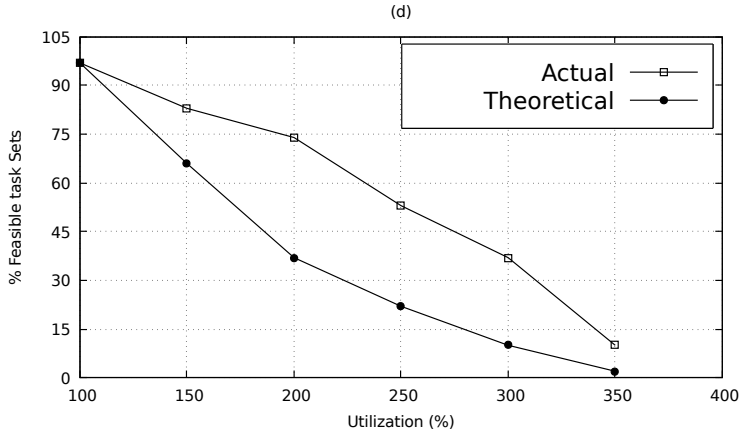
[0]Read as Item[small/short-avg-high/long)]: #TEE tasks[30%−60%−90%], TEE length[30%−50%−70%], Self-suspension length[30% − 50% − 70%]



(a) The percentage of feasible task sets as a function of utilization for use case scenario #1 (Table 9.1)

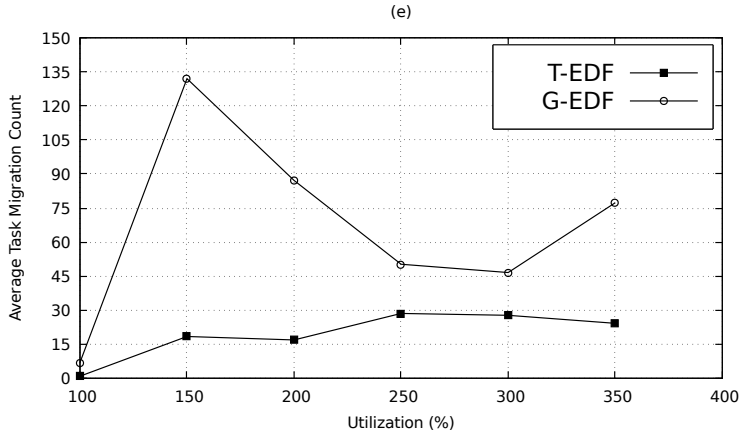


(b) The percentage of feasible task sets as a function of utilization for use case scenario #2 (Table 9.1)

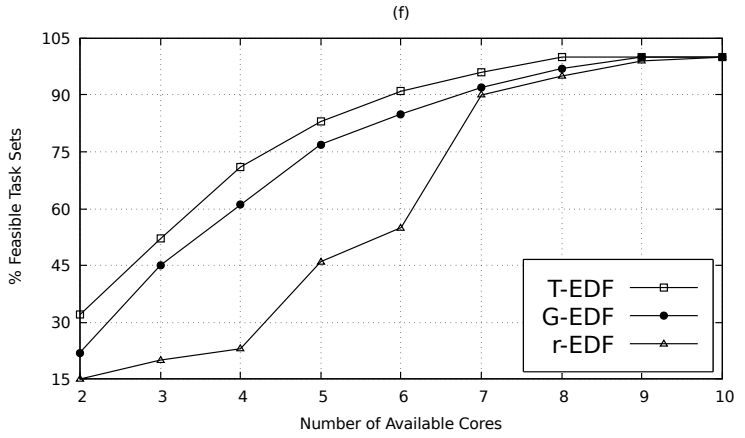


(c) The percentage of feasible task sets of T-EDF as a function of utilization obtained by simulation and feasibility test (use case scenario #2)

Figure 9.2: T-EDF Simulation results



(d) Average task migration count as a function of utilization for T-EDF and G-EDF (use case scenario #2)



(e) The percentage of feasible task sets as a function of core count (use case scenario #2)

Figure 9.3: T-EDF Simulation results cont.

TEE augmentation is small, in both number and percentage of execution time with respect to the rest of the system, the #2 scenario shows the scene where the TEE augmented code is a large percentage of execution and a significant amount of time is spent self-suspended. For each utilization level (100%, 150%, 200%, 250%,..., 400%), we generated 100 task sets of 5 tasks each. For use case scenario #1, each task set comprises of average number of TEE tasks (50% of the tasks in the task set). The trusted execution duration (υ_i) of each TEE task is set at 30% of the worst-case execution time. The self-suspension interval (to account for inter-task data dependency) is also set at 30% of the worst-case execution time. Similarly, for use case scenario #3, each task set comprises of high number of TEE tasks (90% of the tasks in the task set). The trusted execution duration (υ_i) of each TEE task is set at 70% of the worst-case execution time. The self-suspension interval is also set at 70% of the worst-case execution time. We consider a system consisting of 4 processor cores in total. Each simulation is carried out for one hyperperiod, the least common multiple of the periods of all the tasks in a task set. We also test the scalability of T-EDF by varying the core count $(2, 3, 4, \ldots, 10)$.

We now assess the performance of T-EDF. As previously stated, T-EDF aims to tackle three challenges: (i) improve schedulability, (ii) reduce migration overhead, and, (iii) effectively utilize idle processor cycles. Figures 9.2(a)-9.3(e) compare the simulation results to highlight the benefits of T-EDF over G-EDF and r-EDF. Note that both G-EDF and r-EDF are suspension oblivious algorithms. Therefore, we add the self-suspension intervals as part of a task's worst-case execution time for G-EDF and r-EDF. Figure 9.2(a) reports the simulation results comparing the performance of T-EDF against both G-EDF and r-EDF in terms of the percentage of feasible task sets as a function of utilization demands for use case scenario #1 (Table 9.1). The trend shows that G-EDF improves the usable utilization bound by 9% and 26% on average over T-EDF and r-EDF, respectively, across all utilization levels.

T-EDF fails to improve the usable utilization bound set by G-EDF because the tasks do not have enough inter-task data dependency (attributed by short self-suspension intervals) for TEE migrations to exploit optimal resource utilization in T-EDF. For the next set of results we will be using simulation use case scenario #2 (Table 9.1). Figure 9.2(b) reports the simulation results comparing the performance of T-EDF against both G-EDF and r-EDF in terms of the percentage of feasible task sets as a function of utilization demands. The trend shows improved usable utilization bound of 29% and 8% on average over r-EDF and G-EDF, respectively, across all utilization levels. Note that all algorithms policies fail to feasibly schedule task sets once the task utilization demand exceeds 350% utilization. Figure 9.2(c) compares the average performance T-EDF against the worst-case theoretical bounds and confirms its correctness through simulation results. We compare the worst-case percentage of feasible task sets (using the utilization bound) with the actual percentage of feasibly scheduled task sets in simulation. Our results indicate a sufficient and conservative utilization bound which can seamlessly be used to schedule hard real-time task sets. Figure 9.3(d) reports the simulation results comparing the performance of T-EDF against G-EDF in terms of the number of migrations. The trend shows an improvement of 26% on average and up to 86% across all utilization levels. This can be attributed to the design policy of T-EDF which prohibits a task migration to another core unless it is a TEE-task. This is in direct contrast to G-EDF which allows unbounded task migrations. Since r-EDF does not allow any task migration, we skip the comparison between T-EDF and r-EDF.

Figure 9.3(e) reports the simulation results comparing the scalability of T-EDF approach. It compares the performance of T-EDF against both G-EDF and r-EDF in terms of percentage of feasible task sets as a function of number of available cores. We report an average improvement of up to 10% and 50% over G-EDF and r-EDF respectively across all utilization levels.

9.2 Super-TEE and CT-RM

We now consider an orthogonal problem, i.e, space and memory constraints of utilizing TEE. Super-TEE task model and CT-RM scheduling are techniques to improve schedulability by bundling TEEs together.

9.2.1 Overview

As discussed in Section 4, as TEEs are not supported by the same hardware capabilities made available to their rich OS counterparts, TEEs are severely resource constrained. Coupled with simpler OS running inside the TEE, TAs have a smaller range of capabilities as compared to applications running in the rich OS. From a security point of view, this means less attack vectors and higher codebase robustness. However, for real-time applications having periodic tasks that are structured as in Figure 4.1 due to these restrictions, multiple SMC calls to the TEE per period of execution can have a high cumulative overhead.

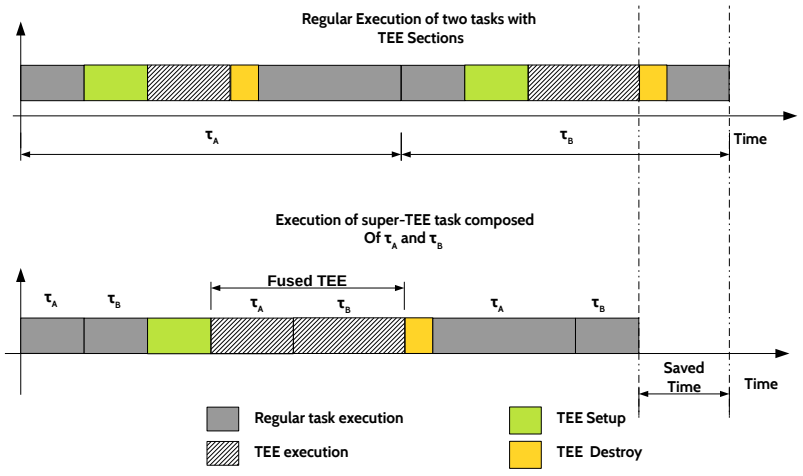


Figure 9.4: A motivating example where (a) two tasks need six SMC calls to access two separate instances of TEE execution, but, (b) fused TEE execution sections reduce the number of SMC calls to three.

To amortize this high cost, the total number of SMC calls should be reduced. This can be made possible by fusing TEE calling tasks together, and calling a fused TA containing the TA code and data required for the original tasks. From Figure 9.4, when two tasks are fused together to create a super-TEE, the advantage is the significant reduction of the the TEE overhead. As is seen, the fused task has one less context and session open and destroy each when compared to the original case . As seen from Table 4.1, the reduction of SMCs results in a significantly reduced cumulative execution time. However, this approach cannot be universally applied due to:

- The cumulative size of the fused TA could be too large for the secure cache and memory.
- The overall system utilization may increase since the super-TEE task must inherit the least period of the original tasks from which it was derived and must include code of the original tasks from which it was derived. This problem is further exacerbated if the execution times and periods of the tasks being fused into the super-TEE are very different from each other.

This approach thus, utilizes three metrics to fuse upto 2 tasks together into a super-TEE. It can be extended to more than 2, but it entirely depends on if the platform would allow a fused TA from 3 or more original TAs. The metrics, in order of their importance, include:

- Ensure the total memory footprint of the fused TA is less than the secure cache size.
- Consider task combinations that yields super-TEEs where the utilization is less than the sum of utilization of the original tasks.
- Higher number of concurrent or overlapping job instances. This ensures that situation such as that seen in Figure 9.4 is seen more frequently. This would maximize the time/number of CPU cycles saved per period of execution of the super-TEE task.

CT-RM is a modification of the fixed priority Rate-Monotonic (RM) real-time scheduling policy built to capitalize on the advantages of the super-TEE task model. It is a partitioned scheduling algorithm, utilizing the first-fit partitioning policy, with two steps:

1. Try first-fit with the sorted list of super-TEE task. Sorting is done with respect to increasing task period.

2. For all super-TEE tasks that cannot be pinned, the individual tasks that the super-TEE is comprised of, are considered separately and pinned using first-fit. This situation could arise because, even though the total utilization of the super-TEE task is guaranteed to be less than sum of that of the individual tasks, the super-TEE task model does have a problem that there is a larger utilization chunk that the needs to be pinned to a core.

Once all tasks are pinned, RM is used to schedule the tasks.

9.2.2 CT-RM Simulations

We validate the CT-RM approach and assess its real-time performance against RM-FF in a simulated environment using randomly generated task sets. Said task sets were generated over a range of utilization levels (100%, 150%, 200%, 250%,..., 500%). For each utilization level, we generated 100 task sets, each of which has a random number of tasks, of which tasks having TEE sections ranges between $30\% - 90\%$. For the reported simulation result, each task set consists of 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 is carried out for one hyperperiod as before.

Figure 9.5 reports the average results comparing the performance of CT-RM against RM-FF

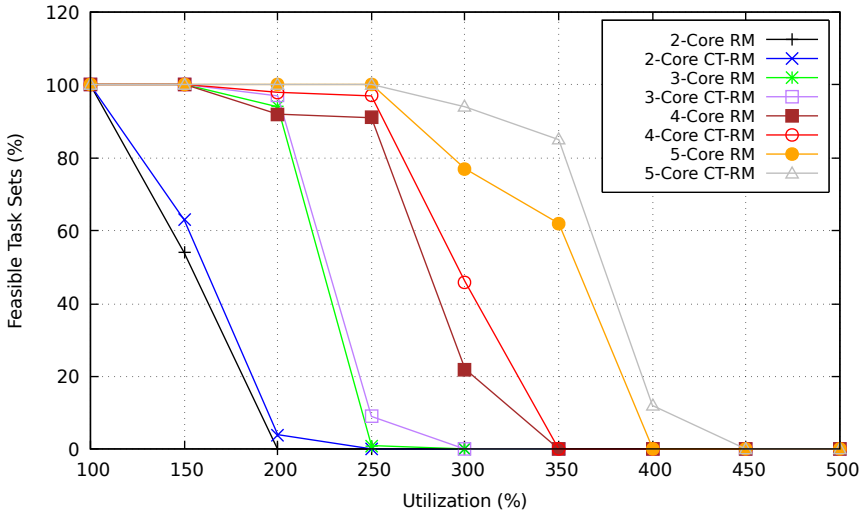


Figure 9.5: Simulation results showing the percent of feasible task sets as a function of utilization levels.

in terms of the number of feasible task sets as a function of utilization levels with scaling cores. The trend shows improved feasibility of up to 73% and 43% on average over partitioned RM-FF across all utilization levels. The region of improvement with our approach (CT-RM) over partitioned RM-FF with scaling utilization levels over increasing core count widens, indicating a scalable solution.

Chapter 10

A hard real-time case study - PX4

We consider the PX4 [3] modular,multithreaded flight stack as a case study for the algorithms presented above.

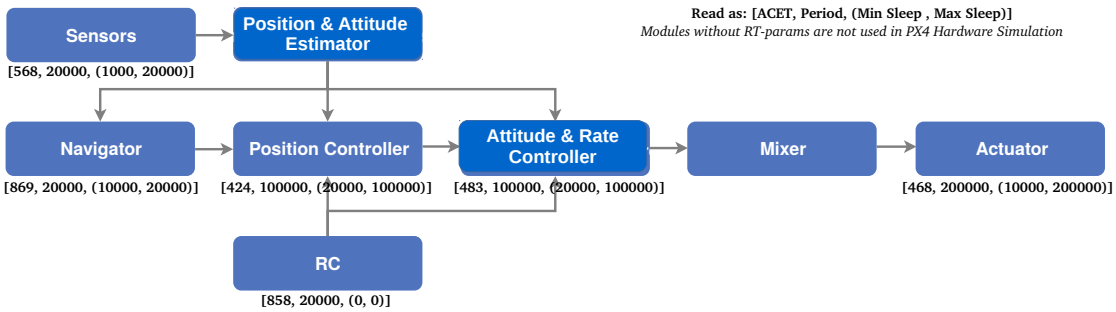


Figure 10.1: PX4 flight stack module intercommunication and RT parameter values on Raspberry Pi 3. All values in μs.

Figure 10.1 shows the different modules that compose the PX4 flight stack and the timing parameters associated with them. Each module runs as a separate thread that is, by default, FCFS (First Come First Served) scheduled. The modules intercommunicate using a shared message buffer called uORB, which has a publish(), subscribe mechanism for modules to broadcast generated data to a subset or all the other modules when generated. The tasks can be modeled as self-suspending periodic task models, very similar to that which T-EDF assumes.

We run the modules with real-time parameters, considering the worst-case situation where

the modules self-suspension times are incorporated into the execution times, with r-EDF, G-EDF and ST-EDF. For all our test cases, we consider that 60% of the tasks have TEE augmentations the TEE augmentation time is rounded out to 18000 μs.

We further simulate running the task set under the constrain that the tasks use only their minimum length self-suspension times, with T-EDF. We consider this situation since this provides the least opportunity for T-EDF to take advantage of its design.

Finally we run the same task set (without self-suspensions) through the super-TEE task model and run it against CT-RM.

In all cases, the task sets fail to meet their deadlines on the standard 4 core system. We consider 4 cores since that is the situation for the Raspberry Pi 3 Model B single board computer and the situation, which is one of the target platforms for this flight stack. The deadline misses are obvious as the TEE overhead itself is nearly 90% of the entire period of the respective tasks. This combined with a synchronous task set where all the tasks are released at the same time and have the same period (harmonic). While ST-EDF allows one more task to complete execution in deadline due to the splitting of the utilization heavy TEE section from the rest of the task, the overall task set fails. Further, the chances of TEE migration for T-EDF is impossible since there is no chance of a self-suspension time and a TEE section overlapping. r-EDF and G-EDF are suspension unaware and fail. CT-RM due to the sheer size of the TEE sections, does not allow any fusing to happen and remains as r-EDF.

The failures showcase that even algorithms smarter than FCFS cannot schedule a hard real-time system augmented with TEE with no consideration given to other real-time parameters. In the perfect world, the algorithm would have been enough for correct and timely scheduling. However, this example shows that not only better scheduling mechanism must be created

for using TEEs, there is a need for re-evaluating the parameters of the hard real-time system when using TEEs.

Chapter 11

A note regarding the security implications of ST-EDF

Before we conclude, we would like to discuss if ST-EDF would inadvertently introduce any security vulnerabilities

11.1 Threat Model

An attacker can convince the system user to run a non-trusted OS userspace application which has permissions to ask for TEE services and could have elevated privileges to run as superuser application. We limit ourselves to a TEE implementation running on the ARM TrustZone, such as OP-TEE.

11.2 Threat Analysis

There exist literature that shows that TEE environments running on ARM TrustZone have vulnerabilities. Some attack techniques utilized elevated privileges on the non-trusted OS, such as the Downgrade Attack [17] exploit the verification procedure for loading trusted application, and the PRIME+COUNT [19] technique which uses ARM cache performance

measurement units as a side channel mechanism respectively. An interesting point to note is that while our mechanism of moving the TEE client API calls into a separate Trusted Application does not change how TEE is being accessed but when it is being accessed, it does not prevent such attacks. However, due to the nature of our algorithm that prefers running the trusted and non-trusted code in separate cores, it reduces the leakage bandwidth for attacks such as PRIME+COUNT which is just 95 B/s for cross-core scenarios versus 27 KB/s for single core scenarios.

When we consider non-elevated privilege model for the attacker there exist attacks such as BOOMERANG [40] where the opaqueness of the memory management unit between the trusted and non-trusted side makes it possible for exploiting the communication channel between trusted applications and the corresponding userspace applications by passing pointers directly to the trusted application, bypassing the pointer sanitization procedure which can especially happen in non-standardized protocol for communicating between TEE and non-trusted OS. ST-EDF remains vulnerable to this technique because this is entirely communication protocol dependent and our scheduling algorithm neither mitigates nor improves this situation.

Finally, moving TEE client API calls to a separate userspace thread does not change the way the TEE is being accessed nor are we changing how the API calls behave. Temporal attacks such as running a higher priority (lower deadline) thread to starve the normal task would have the same effect in case no splitting had occurred because the priority of the normal task would not change instance-to-instance from the original case and the TEE task actually would have a higher priority due to its smaller artificial deadline making it less prone to such attacks.

Chapter 12

Conclusion

We show the applicability of TEE execution to solve the security issues of trust, authenticity and confidentiality in the context of real-time systems. We consider the challenges of using TEEs in real-time systems, considering their significant overhead and performance impact. We present a new task model, the Split-TEE task model, which splits tasks into normal and trusted components, and a new scheduling algorithm ST-EDF. ST-EDF is aimed to improve performance in the worst-case situation for CT-RM and T-EDF. Simulation results show ST-EDF can provide improvement in performance (up to 12%) with an improvement shown as the number of tasks in the system increases. This improvement is almost free as it would require very little code change. However, there is still room for better performance. Our bounds for acceptance are conservative, reducing the feasible taskset space for the ST-EDF algorithm, and must be tightened. Further, there is a need for taskset schedulability test which could deem if a taskset is feasible without simulations, improving the ease of validation of our approach especially for long running tasks. We also provide simulation results for two already proposed algorithms, T-EDF and CT-RM and present their performance improvements. We note that they work as expected in their intended use scenarios and the schedulability bounds reported, though correct, have room for improvement. We also see that the Split-TEE model and ST-EDF can be used as optimizations over the base case of T-EDF and Super-TEE/CT-RM. However, it must be integrated into those algorithms and studied to see the cumulative performance boost that could be possible. We apply

these algorithms to a simulated case study based on the PX4 autopilot flight stack, and note that the scheduling algorithm modifications are not enough and real-time behavior and parameters must be looked into too. Finally, we note that ST-EDF and the Split-TEE task model do not introduce any additional security vulnerabilities over and above the currently documented vulnerabilities.

Bibliography

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

[2] URL http://www.tldp.org/HOWTO/SMP-HOWTO.html.

[3] Pixhawk Autopilot(2005). http://pixhawk.org/. Accessed: 2017-10-06.

[4] Main page. URL https://rt.wiki.kernel.org/index.php/Main_Page.

[5] Widevine. URL http://www.widevine.com/.

[6] A secure web is here to stay, Feb 2018. URL https://blog.chromium.org/2018/02/a-secure-web-is-here-to-stay.html.

[7] Quazi N Ahmed and Susan V Vrbsky. Maintaining security in firm real-time database systems. In Computer Security Applications Conference, 1998. Proceedings. 14th Annual, pages 83–90. IEEE, 1998.

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

[9] T.P. Baker and S.K. Baruah. An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems. Real-Time Systems, 43(1):3–24, September 2009.

[10] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. Real-Time Systems, 4(2):125–144, May 1992.

[11] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. Real-Time Systems, 38(3):223–235, 2008.

[12] Sanjoy K Baruah and John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. Journal of Embedded Computing, 1(2):169–178, 2005.

[13] S.K. Baruah. Scheduling periodic tasks on uniform multiprocessors. In Proc. Euromicro Conf. Real-Time Systems, pages 7–13, June 2000.

[14] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In Proceedings of the conference on Design, automation and test in Europe: Proceedings, pages 15–20. European Design and Automation Association, 2006.

[15] Jian-Jia Chen and Cong Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In Real-Time Systems Symposium (RTSS), 2014 IEEE, pages 149–160. IEEE, 2014.

[16] Jian-Jia Chen, Wen-Hung Huang, and Geoffrey Nelissen. A note on modeling self-suspending time as blocking time in real-time systems. arXiv preprint arXiv:1602.07750, 2016.

[17] Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. Downgrade attack on trustzone. CoRR, abs/1707.05082, 2017.

[18] H. Cho, B. Ravindran, and E.D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In Proc. Real-Time Systems Symp., pages 101–110, December 2006.

[19] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Zim-

ing Zhao, Adam Doupé, and Gail-Joon Ahn. Prime+count: Novel cross-world covert channels on arm trustzone. In ACSAC, 2018.

[20] R.I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys, 43(4):1–44, October 2011.

[21] U.C. Devi. Soft real-time scheduling on multiprocessors. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[22] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the HTTPS certificate ecosystem. In Internet Measurement Conference, 2013.

[23] N. Fisher, J. Goossens, and S. Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. Real-Time Systems, 45(1–2):26–71, June 2010.

[24] Torsten Frenzel, Adam Lackorzynski, Alexander Warg, and Hermann Härtig. Arm trustzone as a virtualization technique in embedded systems. In Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya, 2010.

[25] Binto George and Jayant R. Haritsa. Secure transaction processing in firm real-time database systems. In SIGMOD Conference, 1997.

[26] J Greene. Intel trusted execution technology, white paper. Online: http://www. intel. com/txt, 2012.

[27] Monowar Hasan, Sibin Mohan, Rakesh B Bobba, and Rodolfo Pellizzoni. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In Real-Time Systems Symposium (RTSS), 2016 IEEE, pages 123–134. IEEE, 2016.

[28] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B Bobba. A design-space exploration for allocating security tasks in multicore real-time systems. In Design,

Automation & Test in Europe Conference & Exhibition (DATE), 2018, pages 225–230. IEEE, 2018.

[29] Wen-Hung Huang and Jian-Jia Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In Proceedings of the 2016 Conference on Design, Automation & Test in Europe, pages 1078–1083. EDA Consortium, 2016.

[30] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. SIAM J. Comput., 18:244–257, 1989.

[31] Ke Jiang, Adrian Lifa, Petru Eles, Zebo Peng, and Wei Jiang. Energy-aware design of secure multi-mode real-time embedded systems with fpga co-processors. In Proceedings of the 21st International conference on Real-Time Networks and Systems, pages 109–118. ACM, 2013.

[32] Wei Jiang, Ke Jiang, and Yue Ma. Resource allocation of security-critical tasks with statistically guaranteed energy constraint. In Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on, pages 330–339. IEEE, 2012.

[33] Uri Kanonov and Avishai Wool. Secure containers in android: the samsung knox case study. In Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices, pages 3–12. ACM, 2016.

[34] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys (CSUR), 31(4):406–471, 1999.

[35] Vuk Lesi, Ilija Jovanov, and Miroslav Pajic. Security-aware scheduling of embedded

control tasks. ACM Transactions on Embedded Computing Systems (TECS), 16(5s): 188, 2017.

[36] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM), 20(1):46–61, 1973.

[37] Cong Liu and James H. Anderson. Multiprocessor schedulability analysis for self-suspending task systems ∗. 2011.

[38] Cong Liu and James H. Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. 2013 25th Euromicro Conference on Real-Time Systems, pages 271–281, 2013.

[39] Renju Liu and Mani Srivastava. Protc: Protecting drone's peripherals through arm trustzone. In Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications, pages 1–6. ACM, 2017.

[40] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Krügel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In NDSS, 2017.

[41] Op-Tee. Op-tee/optee_os. URL https://github.com/OP-TEE/optee_os/blob/master/core/arch/arm/plat-rpi3/conf.mk.

[42] Sandro Pinto, Daniel Oliveira, Jorge Pereira, Nuno Cardoso, Mongkol Ekpanyapong, Jorge Cabral, and Adriano Tavares. Towards a lightweight embedded virtualization architecture exploiting arm trustzone. In Emerging Technology and Factory Automation (ETFA), 2014 IEEE, pages 1–4. IEEE, 2014.

[43] Sandro Pinto, Daniel Oliveira, Jorge Pereira, Jorge Cabral, and Adriano Tavares. Freetee: When real-time and security meet. In Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on, pages 1–4. IEEE, 2015.

[44] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. LTZVisor: TrustZone is the Key. In Marko Bertogna, editor, 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), volume 76 of Leibniz International Proceedings in Informatics (LIPIcs), pages 4:1–4:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-037-8. doi: 10.4230/LIPIcs.ECRTS.2017.4. URL http://drops.dagstuhl.de/opus/volltexte/2017/7153.

[45] Concepcio Roig, Ana Ripoll, and Fernando Guirado. A new task graph model for mapping message passing applications. IEEE transactions on Parallel and Distributed Systems, 18(12):1740–1753, 2007.

[46] Bill Rosenblatt, Bill Trippe, Stephen Mooney, et al. Digital rights management. New York, 2002.

[47] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE, pages 71–80. IEEE, 2011.

[48] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. Secpod: a framework for virtualization-based security systems. In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, pages 347–360. USENIX Association, 2015.

[49] Tao Xie and Xiao Qin. Improving security for periodic tasks in embedded systems through scheduling. ACM Trans. Embedded Comput. Syst., 6:20, 2007.