



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Security-aware Scheduling of Mixed-Trust Tasks in Multiprocessor Real-Time Systems

Master's thesis in Computer science and engineering

Jakob Sjöberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

MASTER'S THESIS 2026

Security-aware Scheduling of Mixed-Trust Tasks in Multiprocessor Real-Time Systems

Jakob Sjöberg



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Security-aware Scheduling of Mixed-Trust Tasks in Multiprocessor Real-Time Systems

Jakob Sjöberg

© Jakob Sjöberg, 2026.

Supervisor: Risat Pathan, Department of Computer Science and Engineering
Examiner: Risat Pathan, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2026

Security-aware Scheduling of Mixed-Trust Tasks in Multiprocessor Real-Time Systems

Jakob Sjöberg

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

Real-time systems are a fundamental component of safety-critical cyber-physical systems, such as autonomous vehicles, robotics, and avionics, where missed deadlines can have severe consequences. As these systems become increasingly interconnected, security has emerged as a critical concern, yet enhancing security typically increases computational demands, risking deadline misses. While security mechanisms for uniprocessor real-time systems are well researched, globally scheduled fixed-priority multiprocessor systems have received comparatively little attention. This thesis addresses that gap by introducing the Mixed-Trust task model for globally scheduled fixed-priority multiprocessor systems to enhance security, along with a corresponding schedulability analysis.

The proposed model distinguishes between trusted and untrusted tasks, and upon detection of a security threat, transitions the system to a higher security level by introducing new security enhancing tasks while suspending untrusted ones. Inspired by the work on mixed-criticality scheduling the model additionally supports changing periods for tasks across security levels with the objective to increase the surveillance of the system. Finally to mitigate timing-based attacks, a randomization algorithm extending a uniprocessor algorithm, known as TaskShuffler, to globally scheduled fixed-priority multiprocessor systems is presented, diversifying task execution order to increase unpredictability for a potential attacker. The proposed mechanisms are evaluated empirically: the schedulability analysis is assessed using synthetically generated task sets, revealing how the task model and the number of available processors interact in ways that may help the system designer in selecting the hardware platform. The randomization algorithm is evaluated through simulation, demonstrating a meaningful increase in execution order unpredictability.

Keywords: Real-time Systems, Mixed Criticality, Global Scheduling, Mixed-Trust Security, Multiprocessor TaskShuffler

Acknowledgements

I would like to thank my supervisor, Risat Pathan, for his valuable input and guidance throughout the development of this thesis.

Jakob Sjöberg, Gothenburg, 2026-09-23

Contents

List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Aim and Contributions	4
1.2 Thesis Outline	5
2 Background	7
2.1 Real-time Systems	7
2.1.1 Uniprocessor scheduling	8
2.1.2 Multiprocessor scheduling	9
2.1.3 Optimal Priority Assignment (OPA)	11
2.1.4 Mixed Criticality	12
2.2 Attacker Model	14
2.3 Schedule Randomization	15
3 Related work	19
3.1 Global scheduling and mixed criticality	19
3.2 Security in real-time systems	21
4 Task Model	25
4.1 Scheduling algorithm	28
5 Schedulability analysis	31
5.1 Response-Time Analysis for a Fixed Security Level	32
5.2 Transitioning to a higher security level	34
5.3 Priority assignment and schedulability	41
6 TaskShufflerGlobal: A Randomization Scheduling Algorithm	43
7 Experimental Setup	49
7.1 Task generation	49
7.2 Schedulability Test	51
7.3 Simulation Setup	52
7.4 Runtime Metrics	53

8	Results	55
8.1	Mixed-Trust Model Schedulability	55
8.2	TaskShufflerGlobal	59
9	Conclusion	65
9.1	Future Work	66
	Bibliography	67

List of Figures

2.1	Execution pattern of a periodic task τ_i with parameters C_i , D_i , and T_i .	8
2.2	Partitioned vs global scheduling in a multiprocessor system showing partitioned and shared ready queue structures.	10
2.3	The execution of a high-criticality task executing during a low-criticality level and a criticality switch to a high-criticality level.	13
5.1	The problem window of size t over a level increase.	34
5.2	The job j_k^* contributes C_k^{HS} to the NC workload as the transition happens before $r_k^* + \zeta_k$	36
5.3	The job j_k^* contributes C_k^{NS} to the NC workload as the transition happens after $r_k^* + \zeta_k$	36
5.4	The first release of the high security task τ_k appear at the point s when the level increases.	36
5.5	Release pattern to maximise workload of task τ_k in the problem window of size t when a level increase occur at time $r_i^x + s$	37
5.6	Problem window of size t before a shift	39
5.7	Problem window of size t shifted to the left by $\alpha = d_1 + d_2$	39
8.1	Acceptance ratio of randomly generated task sets composed of 15 tasks, 0.5 HS utilization, and a 1.25 scaling factor, using 2, 4, and 8 processors.	56
8.2	Acceptance ratio of randomly generated task sets with no trusted tasks, 70% trusted tasks, and a fully trusted task set on 4 processors.	56
8.3	Weighted acceptance ratio comparing the effect of trust ratio on schedulability for 2, 4, and 8 processors.	57
8.4	Weighted acceptance ratio comparing how the execution scaling factor of trusted tasks affects schedulability on 2, 4, and 8 processors.	58
8.5	Weighted acceptance ratio comparing how HS utilization affects schedulability on 2, 4, and 8 processors.	59
8.6	Average Task-Level Entropy	60
8.7	Average Processor-Level Entropy of <i>TSG</i> and the <i>TS-BK</i> algorithms on 2, 4, and 8 processors	61
8.8	Average Processor-Level Entropy	61
8.9	Average Processor-Level Entropy comparision of <i>TSG</i> , <i>TS-BK</i> and the non randomized global scheduler using 4 processors	62

8.10 Average entropy of TSG and $TSG+$ using 10 additional tasks execut- ing when normally it should be idle.	63
--	----

List of Tables

2.1	Inversion budgets V_i and minimum inversion priorities M_i for an example task set. Priorities follow task indices, with τ_1 highest and τ_5 lowest.	16
4.1	Example Mixed-Trust task set.	27
6.1	Jobs in the ready queue with their corresponding inversion budgets. .	46

1

Introduction

Real-time systems (RTS) are a fundamental component of cyber-physical systems (CPS), including autonomous vehicles, robotics, and avionics. Meeting timing constraints, such as task deadlines, is essential for ensuring safety in these systems, as missed deadlines can result in severe consequences, including injury to humans and damage to equipment. Traditionally real-time systems have been considered secure since they have not been connected to the internet and physically isolated from the outside world. As CPSs become increasingly interconnected to enable communication and coordination among multiple devices, the risk of malicious activity grows, making security a critical concern for system safety.

Enhancing the system security typically increases computational demands, which may lead to missed deadlines and, consequently, compromised safety. Therefore, developing effective mechanisms that balance strong security guarantees with efficient scheduling is crucial for maintaining both timing guarantees and system safety. Modern high-performance real-time systems increasingly rely on multiprocessor platforms to meet growing computational demands, a trend particularly evident in avionics, automotive, and space applications where multicore platforms help reduce size, weight, and power requirements [1]. This makes it essential that security solutions address multiprocessor environments specifically.

Real-time systems execute independent programs called *tasks* that execute periodically according to a schedule. A commonly used policy for the scheduler is *fixed priority* where the priority of each task is determined offline and is static during runtime. The goal of the scheduler is to successfully execute each task without any deadline being missed as this could compromise the safety of the system.

The schedule of a real-time system is inherently predictable, a property that can be exploited by an attacker with knowledge of the system's behaviour to launch side-channel attacks [2]. An attacker may use such an attack to gain knowledge of the system by observing the timing of task executions. If an attacker has managed to compromise one task, observing the timing of its execution alone may be enough to determine the full system schedule [3]. Once the schedule is known, the attacker may launch precise attacks to gain specific knowledge, such as encryption keys, or to disrupt the system as much as possible.

A related but distinct threat is a covert-channel attack, where two tasks deliberately communicate by manipulating shared resources, such as CPU time, in a way that

affects each other’s execution timing [4]. Unlike a side channel, where information is leaked unintentionally, a covert channel is an intentional signal: a higher-priority task can withhold or claim the CPU in a specific pattern that a lower-priority task observes and decodes. This same predictable scheduling behaviour that enables side-channel attacks may also create the opportunity for covert channels [5].

A method to mitigate both side-channel and covert-channel attacks is schedule randomization. One such method is *TaskShuffler* [5], in which a budget is calculated offline to determine how much each task may be preempted without missing any deadline. During runtime, lower-priority tasks are allowed to preempt higher-priority tasks. By randomizing which task is executing, instead of always executing the highest-priority task, the predictability of the schedule is reduced, making it more difficult for an attacker to predict the schedule or establish a reliable covert channel.

Another method to prevent information leakage is to assign security levels to tasks and limit tasks with different security levels from executing too close together [6]. By flushing shared resources such as the cache between different task executions the information leakage may be prevented at the cost of additional execution time.

While security mechanisms for uniprocessor real-time systems are well established in the literature, the multiprocessor case has received comparatively little attention. Scheduling for multiprocessor systems is a multidimensional problem with issues that are not transferable from the uniprocessor case, which means that security measures developed for uniprocessor systems are not directly transferable either.

Scheduling for multiprocessor systems can be addressed using two different approaches, either *partitioned* or *global* [7]. With the partitioned approach, tasks are assigned offline to different processors, each of which independently determines at runtime which of its assigned tasks to execute. Determining which processor a task is assigned to is equivalent to the bin-packing problem, which is known to be NP-hard. Once tasks have been assigned to a processor, however, the schedulability analysis reduces to m independent uniprocessor scheduling problems, one per processor, allowing the well-understood properties of uniprocessor scheduling to be reused directly.

With the global approach, which is the focus of this work, all tasks are managed by a single shared scheduler and may execute on any available processor. This introduces the possibility of tasks migrating between processors during execution. Furthermore, there is no known scheduling policy for the global case that is optimal. In the uniprocessor case, schedulability may be verified by analysing the *critical instant*: the worst-case scenario in which all tasks arrive simultaneously, producing maximum interference. If the task set is schedulable at the critical instant, it is schedulable under all other conditions. No equivalent critical instant is known for the global multiprocessor case, so this technique cannot be used directly. As a result, task migration and the absence of a known critical instant make schedulability analysis for globally scheduled fixed-priority systems substantially more difficult than in the uniprocessor case.

The technique used in this work to determine whether a globally scheduled fixed-

priority task set is schedulable is Optimal Priority Assignment (OPA) [1]. OPA works by testing task priority orderings against a schedulability test, guaranteeing that if any priority assignment renders the task set schedulable, OPA will find one.

Despite fixed-priority scheduling being widely used in practice, published work addressing security in globally scheduled fixed-priority multiprocessor systems remains limited. [8] extends TaskShuffler [5] to enable schedule randomization on multiprocessor platforms. Other approaches, such as *SecureCore* [9], target dual-core systems but dedicate one core entirely to monitoring and security, reducing the computational capacity available for real-time tasks. While [8] demonstrates that schedule randomization can be extended to multiprocessor systems, it focuses on randomization as an isolated security mechanism rather than as part of a broader security model. These limitations motivate approaches that provide security while making more efficient use of the available computational resources.

Security mechanisms generally introduce additional computational overhead. Additional tasks may be required, schedule randomization may increase the number of context switches, and interactions with co-processors or other hardware components can further increase execution time. To manage this overhead, this work adapts a scheduling principle similar to that used in mixed-criticality scheduling [10].

In the mixed-criticality model, tasks with different levels of importance, or *criticality*, share the same platform. Real-time analysis relies on pessimistic assumptions about a task's worst-case execution time (WCET), which represents an upper bound on the time required for a task to complete. Because higher-criticality tasks require stronger timing guarantees, they are typically analysed using more conservative WCET estimates [11]. The Adaptive Mixed-Criticality model exploits the difference between these estimates by monitoring task execution at runtime. If a task exceeds its lower-criticality execution budget, the system transitions to a higher criticality level and suspends lower-criticality tasks so that higher-criticality tasks can continue to meet their deadlines [12]. This allows additional tasks to execute under typical operating conditions while preserving timing guarantees in the worst case.

A similar principle can be applied to security by grouping tasks according to their level of trust. A method such as Threat Assessment & Remediation Analysis (TARA) [13] could be used to determine the trust levels of tasks, although the model proposed in this thesis does not depend on any specific assessment technique. TARA has been applied in several security-critical domains to evaluate the trustworthiness of software components, for example in connected and automated vehicles [14]. When a security threat is detected, untrusted tasks can be suspended to free computational resources for additional security mechanisms while preserving the execution of essential trusted tasks. Untrusted tasks are permitted because they may provide beneficial functionality to the system, but they may not be certified or assessed to the same level of trust as the other system tasks.

1.1 Aim and Contributions

The aim of this work is to develop a security-aware task model for global fixed-priority real-time systems that integrates tasks with different trust levels and security functionality. This aim is addressed through the following goals:

- The first goal is to develop a security-aware task model that enables a real-time system to increase its security response when an attack is detected while managing the computational resources available to the system. This is challenging because security functionality introduces additional activities, such as monitoring for threats and responding to detected attacks, that must execute alongside application tasks. Furthermore, the system must be able to change its execution behaviour as its security level changes while continuing to satisfy its real-time requirements. The contribution of this work is the *Mixed-Trust Task Model*, a task model that distinguishes between trusted and untrusted tasks and integrates them with security functionality within a unified model. A *Mixed-Trust Scheduler* is proposed for the model, supporting security-based level switching on a global fixed-priority multiprocessor system and allowing the system to change its execution behaviour in response to changes in its security level.
- The second goal is to establish the schedulability of systems using the proposed *Mixed-Trust Task Model* under global fixed-priority scheduling. This is challenging because security-based level switching can change the timing behaviour of the system, including task periods and the set of tasks executing at higher security levels. These changes must be accounted for when determining whether the system can satisfy its real-time requirements. The contribution of this work is a response-time analysis that supports level-dependent execution times and periods, as well as the introduction of additional tasks at higher security levels. The analysis is applicable both to the proposed mixed-trust model and to mixed-criticality systems exhibiting similar behaviour.
- The third goal is to reduce the predictability of task execution in global fixed-priority systems in order to improve resilience against attacks that exploit predictable scheduling behaviour. This is challenging because introducing randomness into task execution must be achieved without compromising the scheduling and timing requirements of the system. The contribution of this work is the *TaskShufflerGlobal Algorithm*, a global fixed-priority implementation of TaskShuffler that introduces randomized task execution to reduce schedule predictability and improve resilience against schedule-based attacks.
- The fourth goal is to experimentally evaluate the proposed security-aware task model, scheduling mechanisms, response-time analysis, and task execution randomization. This is challenging because the proposed approaches must be evaluated under representative system configurations while considering both real-time performance and security-related objectives. The contribution of this work is an experimental evaluation consisting of two parts. The first evaluates the proposed response-time analysis, while the second evaluates the task execu-

tion randomization and compares the *TaskShufflerGlobal Algorithm* with the previous multiprocessor implementation of task execution randomization [8]. Schedule entropy is used to evaluate the effectiveness of task execution randomization. The evaluation considers both the established schedule entropy metric used in [5] and a new metric that accounts for the use of multiple processors. The results demonstrate an improvement in schedule entropy for *TaskShufflerGlobal* compared with the previous multiprocessor implementation.

1.2 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 introduces relevant background on real-time systems, including uniprocessor and multiprocessor scheduling. Key analysis techniques, such as response-time analysis and priority assignment using the OPA algorithm, are also presented.

Chapter 3 provides an overview of related work in multiprocessor scheduling and security in real-time systems.

Chapter 4 introduces the Mixed-Trust Task Model, its components, and the proposed scheduling algorithm.

Chapter 5 presents the response-time analysis of the proposed scheduler.

Chapter 6 introduces the TaskShufflerGlobal algorithm for schedule randomization.

Chapter 7 describes the experimental setup for both the schedulability analysis and the simulation of the randomization algorithm.

Chapter 8 presents and discusses the results of the experiments.

Finally, Chapter 9 concludes the thesis and outlines directions for future work.

2

Background

This chapter provides the theoretical foundation for the task model developed in this thesis. It introduces the fundamentals of real-time systems, explains the principles of mixed-criticality analysis, describes the randomization mechanisms, and outlines the schedule-based attacks that motivate the need for enhanced security measures. Together, these concepts establish the background required to understand the design and analysis of the proposed model.

2.1 Real-time Systems

A real-time system (RTS) consists of one or more identical processors that execute a set Γ of n tasks, each subject to specified timing constraints. Each task, $\tau_i \in \Gamma$, is an independent computation characterized by a relative deadline D_i , which denotes the latest time by which a job of the task must finish after its release.

Tasks are typically classified as either periodic or sporadic. A periodic task is released at regular intervals, whereas a sporadic task is released with a minimum inter-arrival time between consecutive releases. This minimum interval is referred to as the task period T_i .

The worst-case execution time (WCET), denoted by C_i , represents the maximum amount of processor time required for any instance of task τ_i to complete.

A task can be characterized by the tuple:

$$\tau_i = \{C_i, D_i, T_i\} \tag{2.1}$$

An instance of a task is called a job. Instance x of task τ_i is denoted by j_i^x , and its release time is denoted by r_i^x . Upon release, the job enters the ready queue. The scheduler selects jobs from the ready queue for execution on the available processor(s) according to the employed scheduling algorithm.

Figure 2.1 illustrates the execution of a periodic task $\tau_i = \{C_i, D_i, T_i\}$ over time. Each upward arrow represents a job release of task τ_i , with the interval between consecutive releases equal to the task period T_i . Each downward arrow represents the job deadline, while each blue box indicates the execution time C_i .

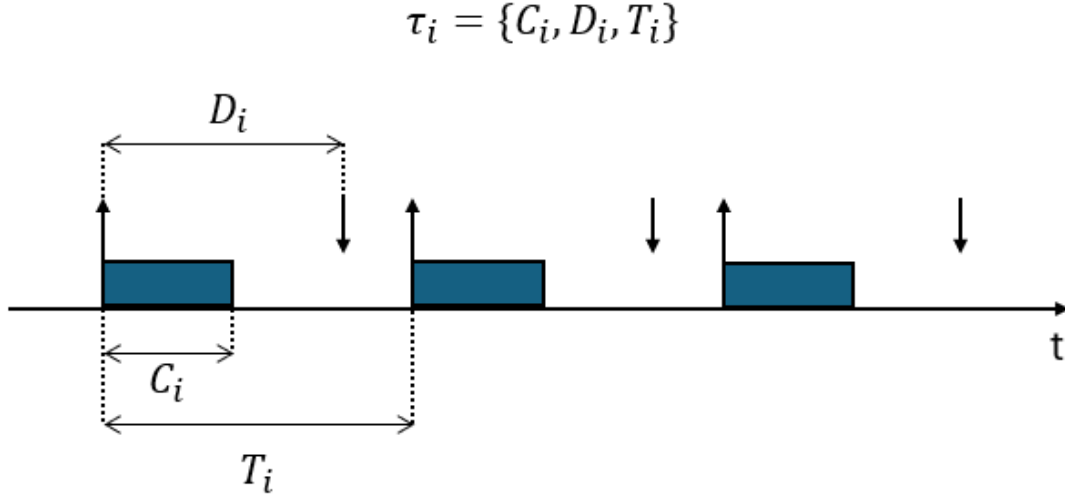


Figure 2.1: Execution pattern of a periodic task τ_i with parameters C_i , D_i , and T_i .

The following sections describe how a set of tasks is scheduled and analysed on both uniprocessor and multiprocessor platforms. Section 2.1.1 introduces uniprocessor scheduling, followed by multiprocessor scheduling in Section 2.1.2.

2.1.1 Uniprocessor scheduling

Scheduling determines the execution order of ready jobs on the processor. This decision is governed by a scheduling policy, which assigns priorities at either the task level or the job level. Priorities may be either fixed, meaning they are assigned once and never change, or dynamic, meaning they may change during runtime. With preemption allowed, a higher-priority job may preempt a lower-priority job as soon as it becomes ready for execution.

Under fixed-priority (FP) scheduling, a priority is assigned to each task, and all job instances of that task inherit the same priority. In contrast, dynamic-priority scheduling assigns priorities to jobs at runtime. A common example is earliest deadline first (EDF), where each ready job is assigned a priority based on its absolute deadline. If all tasks meet their timing constraints under a given scheduling policy the taskset is said to be schedulable. In this work, hard real-time systems are considered where deadline misses are not permitted.

A scheduling policy is considered optimal if it can successfully schedule any task set that is schedulable by any other policy under the same system model. For pre-emptive uniprocessor systems with constrained deadlines ($D_i \leq T_i$), the deadline monotonic (DM) policy, where tasks with shorter deadlines are assigned higher priorities, is optimal among fixed-priority preemptive scheduling policies [15]. Similarly, EDF is optimal among dynamic-priority scheduling policies for preemptive uniprocessor systems [16]. In this work, only fixed-priority scheduling is considered in the

schedulability analysis.

A feasibility test is used to determine whether a task set is schedulable under a given scheduler. The three types of tests are:

- **Sufficient test:** If the test is passed, the task set is guaranteed to be schedulable. However, a failed test does not imply that the task set is unschedulable.
- **Necessary test:** If the test is failed, the task set is guaranteed to be unschedulable. However, a passed test does not guarantee that the task set is schedulable.
- **Exact test:** A test that is both necessary and sufficient. A task set is schedulable if and only if it passes the test.

Different feasibility tests have different computational complexity. In most cases exact schedulability tests are computationally expensive, NP-complete problems or have exponential time complexity. However, under additional assumptions such as implicit deadlines ($D_i = T_i$) or restrictions on resource sharing, some tests can be evaluated in polynomial time.

Response-time analysis (RTA) is an offline method for evaluating schedulability under fixed-priority scheduling [17]. RTA is an exact feasibility test that can be run with pseudo-polynomial complexity. The response time of a job is defined as the time interval between its release and its completion. The analysis computes the worst-case response time of each task τ_i by accounting for interference from higher-priority tasks, denoted $hp(i)$. A task set is deemed schedulable if the computed response time of every task does not exceed its deadline.

The worst-case response time R_i of a task τ_i is computed iteratively as:

$$R_i^{k+1} = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \quad (2.2)$$

where C_i is the worst-case execution time of task τ_i , T_j is the period of the higher-priority task τ_j , and $hp(i)$ denotes the set of tasks with higher priority than τ_i .

The iteration starts with:

$$R_i^0 = C_i \quad (2.3)$$

and continues until convergence, i.e., $R_i^{k+1} = R_i^k$, or until $R_i^k > D_i$, in which case the task is unschedulable.

A task set is schedulable if and only if:

$$R_i \leq D_i \quad \forall \tau_i \in \Gamma \quad (2.4)$$

2.1.2 Multiprocessor scheduling

In multiprocessor platforms, tasks can be assigned to processors using either partitioned or global scheduling strategies [18]. In partitioned scheduling, each task is statically assigned to a specific processor and always executes on that processor,

in essence multiple uniprocessor systems. In contrast, this work considers global scheduling, where tasks are not statically bound to a processor and may migrate between processors during execution. However, each job is restricted to execute on at most one processor at a time.

Figure 2.2 illustrates the difference between the two approaches. In the partitioned case, each processor μ has its own ready queue containing the tasks assigned to it. In the global case, there is a single shared ready queue from which any available processor can select the next job for execution.

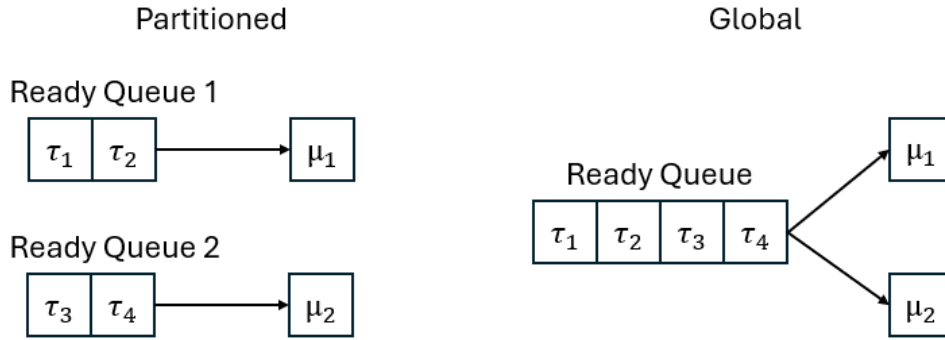


Figure 2.2: Partitioned vs global scheduling in a multiprocessor system showing partitioned and shared ready queue structures.

Global scheduling can improve processor utilization through better load balancing. For example, if jobs are ready for execution, the scheduler can assign them to any available processor. In contrast, under partitioned scheduling, one processor may remain idle even though another processor has a backlog of ready jobs. However, global scheduling may introduce increased preemption and migration overhead compared to the partitioned scheduler.

Determining schedulability under global scheduling is strongly NP-Hard. When tasks can execute on many processors multiple difficult scenarios make the analysis difficult. One is the so called Dhall's effect that when trying to schedule a multiprocessor system using DM that is optimal for uniprocessors a task set can make a taskset unschedulable regardless of how many processors are used [19]. A single task with a large utilization demand in the system can miss deadlines if the global scheduler does not prioritise larger tasks. Additionally no static priority assignment of task priorities can guarantee any task set to be scheduled with a utilization that is more than 50% of the total platform capacity [20]. This is true for both partitioned and global scheduling approaches.

When analysing FP uniprocessors it is enough to analyse a point called the critical instant when the response time of a task is maximized. For uniprocessors this is when the task begin analysed arrives at the same time as all other tasks with a higher priority. If the task set is schedulable at the critical instant it is schedulable

at all other time instances as well. In the multiprocessor case the critical instant does not necessarily occur when a task arrives at the same time as all the higher priority tasks. Instead this approach would be exponential as all possible instances in the schedule would need to be analysed. RTA can be used for multiprocessor but is no longer exact, instead only sufficient, as the critical instant is not known it is more pessimistic than in the uniprocessor case.

No optimal scheduling policy is known for the general case for global scheduling. However, methods such as the Optimal Priority Assignment (OPA)[21] algorithm can be used to assign priorities and assess schedulability of multiprocessor systems [1]. With proper priority assignment issues like Dhall's effect can be mitigated.

2.1.3 Optimal Priority Assignment (OPA)

When a specific task model does not have an optimal scheduling policy, scheduling a RTS becomes a two-step process: first assigning task priorities and then checking the feasibility of the task set. A naive approach would be to evaluate every possible priority assignment of a task set, requiring consideration of all $n!$ possible priority orderings. However, the OPA algorithm provides a method for assigning priorities in polynomial time [21].

The OPA algorithm takes as input a set of tasks without predefined priorities and assigns priorities based on a given schedulability test. Starting from the lowest priority level, the algorithm selects an unassigned task and applies the schedulability test, assuming that all remaining unassigned tasks have higher priorities. If the test succeeds, the task is assigned the current priority level, and the algorithm proceeds to the next priority level. If the test fails, another unassigned task is selected and evaluated under the same assumption.

If all tasks are successfully assigned priorities, the algorithm returns success, indicating that the task set is schedulable under the given schedulability test. However, if no task can be assigned at a particular priority level, the algorithm returns failure, meaning that the task set cannot be scheduled according to the given test under any priority assignment. If the schedulability test is not exact, a different schedulability test may still classify the same task set as schedulable.

For a task set with n tasks, OPA performs at most $(n^2 + n)/2$ schedulability tests. If a valid priority assignment exists under the given schedulability test, the algorithm is guaranteed to find a priority ordering that makes the task set schedulable. Although the priority assignment procedure has polynomial complexity, the schedulability test itself may have non-polynomial complexity, and the effectiveness of the overall approach depends strongly on the accuracy and efficiency of the chosen test.

Pseudocode for the OPA algorithm is provided below:

```
1  OPA(Taskset)
2      for each priority level p lowest first:
3          for each unassigned task t in Taskset:
4              if (t is schedulable with priority p):
5                  t.priority = p
6                  break to outer loop
7          return failure //no task was schedulable at current p
8      return success
```

When considering global scheduling there exists task sets where the response time of a task does not only depend on the WCET and period of higher priority tasks, but also on the relative priority ordering of those tasks [22]. To be able to use a schedulability test with OPA on a multiprocessor system 3 conditions of the test needs to be fulfilled [1]. The conditions refer to properties of tasks in the task set. Properties that are independent are for example WCET, deadline, and period while response time is a dependent property as it is affected by the priority of the task. The conditions for the schedulability test are as follows:

- C1 The schedulability of a task τ_i according to the test may only depend on independent properties of any tasks with a higher priority than τ_i , not on any that depend on the relative priority ordering of the tasks.
- C2 The schedulability of a task τ_i according to the test may only depend on independent properties of any tasks with a lower priority than τ_i , not on any that depend on the relative priority ordering of the tasks.
- C3 If adjacent priority levels between 2 tasks are swapped the task assigned a higher priority cannot be deemed unschedulable with a higher priority if it was previously schedulable according to the schedulability test.

2.1.4 Mixed Criticality

Many safety-critical real-time systems are subject to certification requirements that rely on highly pessimistic assumptions about task execution times. These worst-case execution time estimates are often significantly higher than the time a task typically requires in practice, and not all tasks in a system have equal importance. This distinction between required assurance levels is reflected in industry standards such as ISO 26262 for automotive systems [23] and DO178C for avionics software [24]. ISO 26262 defines Automotive Safety Integrity Levels (ASILs) from A to D, while DO178C defines Development Assurance Levels (DALs) from A to E. Higher levels require more rigorous safety and verification measures, as failures at these levels can have serious consequences.

Mixed-criticality scheduling theory models these differing assurance requirements by assigning tasks criticality levels that correspond to such industry classifications. Tasks of higher criticality are associated with more conservative WCET assumptions, while lower-criticality tasks may rely on less pessimistic estimates. The model in

[11] formalizes this by giving each task multiple WCET values and defining system behaviour when operating at higher criticality levels: lowercriticality tasks may be degraded or suspended to ensure that highly critical tasks can meet their deadlines under stricter timing assumptions.

A mixed criticality (MC) system that is globally scheduled using fixed priority tasks was presented in [10]. The system model consist of a set Γ of n tasks $\{\tau_1, \dots, \tau_n\}$. Each task τ_i is characterized by a tuple (L_i, C_i, D_i, T_i) where L_i is the criticality level, D_i is the relative deadline, T_i is the task period, and C_i is a vector of WCETs, one for each criticality level in the system. Considering two criticality levels, LO and HI, it is assumed that $C_i^{LO} \leq C_i^{HI}$. The system starts in the LO criticality level and each job is monitored to check if the execution exceeds the WCET of the current criticality level. If any job exceeds the WCET the system transitions to the HI criticality level, any task with a lower criticality level will abort any currently executing jobs and no new jobs will be released.

Figure 2.3 illustrates the execution pattern of a HI-criticality task $\tau_i = \{C_i^{LO}, C_i^{HI}, D_i, T_i\}$ before and after a criticality switch. The task initially executes under the LO-criticality assumption using C_i^{LO} , but after a criticality switch it continues execution using the high-criticality WCET C_i^{HI} . The switch is triggered when the task exceeds the optimistic assumption that its execution would complete within C_i^{LO} .

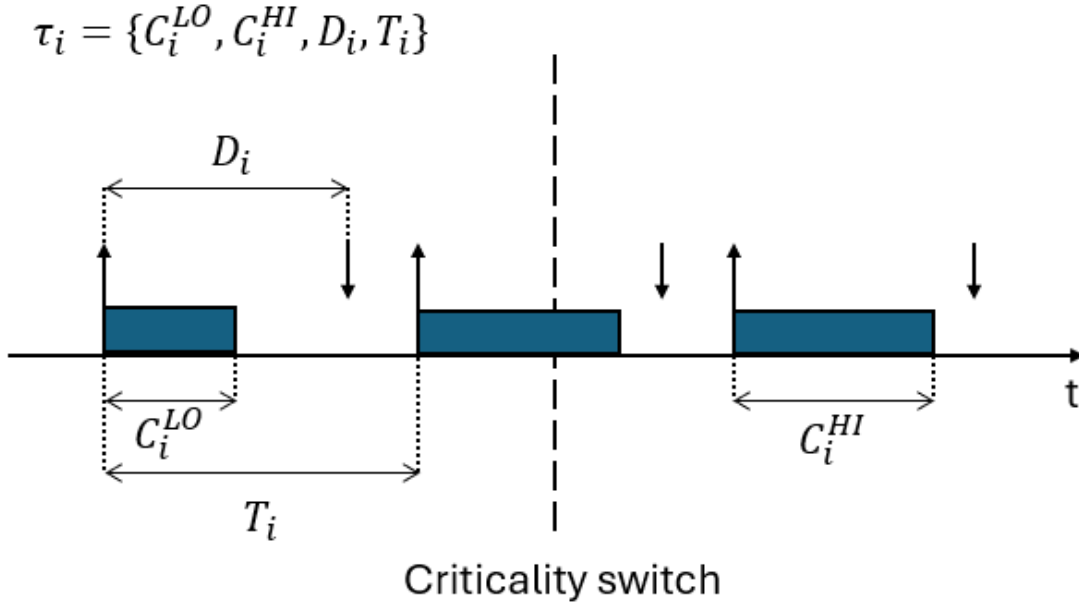


Figure 2.3: The execution of a high-criticality task executing during a low-criticality level and a criticality switch to a high-criticality level.

One way to determine the schedulability of a mixed-criticality task set is to apply response-time analysis (RTA) at each criticality level and at any transition between levels.

The response time of a task in a mixed-criticality system is determined by its *workload*, the *interfering workload* of higher-priority tasks, the *total interfering workload*,

and the resulting *interference* over an interval of length t , referred to as the problem window [10]. These quantities are defined as follows:

- The *workload* of a task τ_k is the cumulative execution time contributed by τ_k within the problem window.
- The *interfering workload* of a higher-priority task $\tau_k \in hp(i)$ is the amount of execution that delays τ_i while it is ready to execute. This interference can originate from either carry-in (CI) or non-carry-in (NC) jobs. CI interference occurs when a job of τ_k released before τ_i becomes ready is still executing at that time, whereas NC interference is caused by jobs released at or after the release of τ_i .
- The *total interfering workload* is the sum of the interfering workloads contributed by all tasks in $hp(i)$.
- The resulting *interference* is the time interval during which τ_i is ready but cannot execute due to higher-priority activity.

At the LO-criticality level, all tasks execute and the total interference is computed using C_i^{LO} . At the HI-criticality level, all LO-criticality tasks are suspended and therefore do not contribute to the RTA, while HI-criticality tasks may have larger WCETs, requiring a separate analysis. For a transition from LO to HI criticality, the RTA must account for the execution of LO-criticality tasks up to the switch, after which they are suspended, as well as the increased execution time of HI-criticality tasks to C_i^{HI} after the transition.

2.2 Attacker Model

Real-time systems are vulnerable to side-channel attacks due to the predictability of their schedules [3]. The hyperperiod of a task set is defined as the least common multiple (LCM) of all task periods. Once the hyperperiod has elapsed, the schedule returns to its initial state, causing the execution pattern to repeat. As a result, the release and execution behaviour of all tasks becomes predictable across hyperperiods. For example, the highest-priority tasks released at system startup will always execute in the first time slot of each hyperperiod, while the execution of all other tasks follows the same deterministic pattern. This predictability enables attackers to anticipate when specific tasks will execute. Four classes of schedule-based attacks are described in [25], where the *victim* is the targeted task:

- **Posterior attack** performed after the victim task to manipulate its output.
- **Anterior attack** performed before the victim task to inject false or misleading input data.
- **Pincer attack** performed both before and after the victim task, often to observe state changes such as cache effects.
- **Concurrent attack** performed at the same time as the victim task to exploit shared hardware or software resources.

The anterior attack has been shown to be effective in disturbing the steering commands in a remote controlled vehicle [3]. By injecting data at the point immediately after the task responsible for steering the attack was able to stop the wheels from turning. Real world implications of this could be a crash. A pincer attack was used to be able to determine the location of important locations by observing the cache before and after an imaging task performed by a drone [3].

The goal of the attacker considered in this work is to exploit predictable execution behaviour in the global fixedpriority schedule in order to disrupt task execution or infer sensitive information. We assume that the attacker has knowledge of the systems task parameters and scheduling policy. By observing the system over time, the attacker attempts to identify execution windows in which an attack would be most effective. One approach is for the attacker to compromise a task and monitor its own execution behaviour. Prior work has shown that observing the execution of a single lowpriority task may be used to reconstruct the full schedule in fixedpriority uniprocessor systems [2]. In the worst-case scenario considered, the attacker is able to infer the exact schedule with high accuracy and thus determine precisely when a timing-based attack would have maximal impact.

2.3 Schedule Randomization

TaskShuffler [5] is an algorithm that addresses the predictability of real-time systems by introducing schedule randomization, allowing lower priority tasks to preempt higher priority tasks in a fixed priority uniprocessor system. To control this safely, the algorithm introduces the worst-case inversion budget V_i for each task τ_i . This budget represents the available slack between the task deadline and the sum of its execution time and interference I_i from higher-priority tasks. Formally, it is defined as:

$$V_i = D_i - (C_i + I_i) = D_i - \left(C_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{D_i}{T_j} \right\rceil + 1 \right) C_j \right) \quad (2.5)$$

A positive inversion budget indicates that the schedule can allow lower-priority tasks in the ready queue to execute instead of higher-priority tasks, thereby introducing run-time randomization.

In addition to the worst-case inversion budget, the minimum inversion priority M_i is an important parameter. It determines the lowest-priority task that is allowed to preempt τ_i without causing another task with a non-positive budget to miss its deadline. This ensures the schedulability of tasks that may not currently have any jobs in the ready queue but are still affected by earlier preemptions when they eventually become ready.

For a task τ_i , M_i is determined by examining all lower-priority tasks $lp(i)$ and identifying the highest-priority task among them with $V \leq 0$. This task defines the minimum inversion priority for τ_i . Table 2.1 shows an example task set with V_i and M_i , where priorities correspond to the task index (1 being the highest and 5 the lowest). Tasks τ_1 and τ_2 share the same $M_i = \tau_3$, as τ_3 is the next task in priority

order with non-positive V_i . Similarly, τ_3 and τ_4 share $M_i = \tau_5$, which is the next task with non-positive V_i . The implication of this is that τ_4 and τ_5 are not allowed to execute if any of τ_1, τ_2 , or τ_3 have unfinished jobs.

Task	V_i	M_i
τ_1	2	τ_3
τ_2	1	τ_3
τ_3	0	τ_5
τ_4	2	τ_5
τ_5	0	-

Table 2.1: Inversion budgets V_i and minimum inversion priorities M_i for an example task set. Priorities follow task indices, with τ_1 highest and τ_5 lowest.

Schedule entropy is a measure of the unpredictability of a system using the TaskShuffler algorithm [5], based on Shannon entropy [26]. A small value indicates that tasks execute in a predictable manner, typically in the same time slots, while a large value means that task executions are widely distributed and the system appears more random. The schedule entropy for a task set Γ is defined as:

$$H_\Gamma(\mathcal{S}) = \sum_{t=0}^{L-1} H_\Gamma(S_t) \quad (2.6)$$

where the sum is taken over the hyperperiod L , and $H_\Gamma(S_t)$ is the slot entropy, measuring the probability $\Pr(s_t)$ of a task executing in time slot t . Slot entropy is defined as:

$$H_\Gamma(S_t) = - \sum_{s_t=1}^N \Pr(s_t) \log_2 \Pr(s_t) \quad (2.7)$$

The probability $\Pr(s_t)$ can be estimated by simulating the execution of the task set over multiple hyperperiods and counting how often each task executes in time slot t .

Intuitively, schedule entropy reflects how much uncertainty an observer faces when trying to predict which task executes in a given time slot. If a task consistently executes at the same time, its probability $\Pr(s_t)$ is concentrated in a single slot, yielding low entropy. When executions are spread across many slots with similar probability, the distribution becomes more uniform and entropy increases. Higher entropy therefore corresponds to a schedule that is harder to predict and offers greater resistance to timing-based side-channel attacks.

The work in [8] extends the TaskShuffler algorithm to global multiprocessor scheduling. The inversion budget is changed as follows to account for multiple processors:

$$V_i = D_i - \left(C_i + \frac{I_i}{m} \right) \quad (2.8)$$

No minimum inversion priority is defined for each task in [8]. In the multiprocessor case, the schedulability analysis already accounts for the worst-case interference patterns, making an explicit M_i parameter unnecessary. Instead, at each scheduling decision, the highestpriority task with a ready job and a nonpositive inversion budget (V_i0) is the one that cannot be preempted by any lowerpriority job. This task may change dynamically as budgets evolve across scheduling points.

An algorithm is provided that maintains a list of candidate jobs, LC , from which jobs will be picked to execute on m processors [8]. The algorithm iterates through all jobs in the ready queue in priority order, from highest to lowest, and adds them to the candidate list until it encounters a job with $V_i \leq 0$ (lines 4–7). At that point, the algorithm stops and checks whether at least m jobs have already been added to LC (line 9). If not, the remaining positions are filled with the highest-priority jobs still available until m candidates have been selected (line 10). Otherwise, the algorithm randomly selects m jobs from LC to execute (line 12). The pseudocode below presents an adapted version of the algorithm in [8], assuming that at least $m + 1$ jobs are present in the ready queue. If fewer than $m + 1$ jobs are available, the algorithm is not executed and no randomization is performed.

```

1 TaskShuffler for multiprocessors:
2   LC = []
3
4   for job in readyQ:
5       LC.append(job)
6       if job.V <= 0:
7           break
8
9   if |LC| < m:
10      return top m jobs by priority from readyQ
11  else:
12      return m jobs randomly from LC

```

Functionally, this algorithm does not randomize if any of the m highest-priority jobs have a non-positive budget. This may be more conservative than necessary. For example, on an eight-processor system, if only the highest-priority job has $V_i \leq 0$, that job must execute, but the remaining seven processors could still be used for randomized execution of other tasks. The algorithm also does not allow a processor to remain idle when budgets permit it. As shown in [5], allowing idle time slots can significantly increase schedule entropy. Furthermore, [8] always executes a ready job, even if it has positive budget. This means that at every job release some job will execute, and at low utilization levels many positive budgets will never be used because tasks begin execution immediately when they become ready. In this work, a new algorithm is proposed which improves the randomization on globally scheduled multiprocessor systems.

The metric work in [8] use the same schedule entropy for the multiprocessor case as the one used for uniprocessors. A limitation of this metric is that it does not distinguish between processors; it only considers the time slots in which tasks are

2. Background

scheduled. Since processors may not share all resources, the *L1* cache for example is typically private, measuring only the time slot may not be sufficient. Since an important property of a global scheduler is that tasks can migrate the processor executing a task should also be reflected in the metric which will be explored in this work.

3

Related work

This chapter surveys work related to the two core aspects of this thesis. The first concerns real-time scheduling, covering global fixed-priority multiprocessor scheduling and mixed-criticality systems. The second concerns security in real-time systems, including both uniprocessor and multiprocessor approaches.

3.1 Global scheduling and mixed criticality

Global fixed-priority multiprocessor scheduling has been studied extensively. Early work extended rate-monotonic scheduling from uniprocessors [20], introducing the RM-US[$m/(3m - 2)$] algorithm, which guarantees schedulability for any periodic task set with total utilization $U \leq m/3$ on m identical processors. By assigning high priorities to tasks with $U \geq \frac{m}{3m-2}$ and otherwise using rate-monotonic priorities, the algorithm mitigates the behaviour associated with Dhall's effect. This work also showed that no static-priority multiprocessor scheduler, partitioned or global, can achieve a utilization bound exceeding 50% of platform capacity.

Response-time analysis for global fixed-priority scheduling was significantly advanced in [7]. The analysis applies to any work-conserving global scheduler, meaning any scheduler that keeps processors busy whenever ready jobs exist. The work addresses the absence of a clear critical-instant definition in multiprocessor systems by using the concept of a busy window and deriving an upper bound on the interference a task may experience, resulting in a sufficient schedulability test for multiprocessor platforms.

Subsequent refinements further tightened the schedulability bounds by distinguishing carry-in and non-carry-in interference within a busy window [27]. By limiting the upper bound on interference to at most $m - 1$ higher-priority carry-in tasks, rather than all higher-priority tasks, the pessimism in the analysis is significantly reduced.

Priority assignment for global fixed-priority multiprocessor scheduling was improved by extending Optimal Priority Assignment (OPA) to multiprocessor platforms [1]. This work defined the conditions under which a schedulability test is compatible with OPA in the multiprocessor setting and demonstrated the substantial impact that priority assignment has on the number of schedulable task sets under global fixed-priority scheduling.

The foundation for mixed-criticality scheduling was established by [11] for uniprocessor systems, providing a framework that allows tasks with varying design assurance levels to share a single platform. Four assurance levels are used (A through D) corresponding to criticality levels in avionics, with Level A representing the highest criticality where erroneous behavior can lead to severe consequences, such as the loss of an aircraft and level D being suboptimal or inconvenient. A central contribution of this work is the conjecture that higher degrees of assurance require increasingly conservative and larger WCET parameters. To analyse these systems, the paper extends traditional Response Time Analysis by calculating response times using the execution time bound that matches the required assurance level for each task.

The adaptive mixed-criticality (AMC) model was introduced in [12]. Under this model two levels of criticality is defined, LO and HI, and each task is assigned to one of them. HI-criticality tasks are characterized by two WCET estimates, C_{LO} and C_{HI} , where $C_{LO} \leq C_{HI}$. The system has a level indicator that initially starts at the LO-criticality level. During runtime the execution of every job is monitored, if a HI-criticality task exceeds its C_{LO} execution time without signalling completion, the indicator switches to the HI-criticality level. Following this change, all LO-criticality tasks are suspended and are no longer eligible for execution, allowing processor resources to be dedicated to HI-criticality tasks.

Schedulability analysis under the AMC model requires verifying the following three conditions:

- The task set is schedulable at the LO-criticality level.
- The HI-criticality tasks are schedulable at the HI-criticality level.
- The task set remains schedulable during the transition from the LO-criticality level to the HI-criticality level.

Mixed-criticality scheduling was subsequently extended to globally scheduled fixed-priority (FP) multiprocessor systems in [10]. This work introduced the Mixed-criticality Scheduling on Multiprocessors (MSM) algorithm, together with a schedulability analysis for globally scheduled FP multiprocessor systems. MSM adopts the same level indicator and task execution monitoring concepts as the AMC model, adapted to account for the behavior of a global scheduler. The analysis is based on response-time analysis for global fixed-priority scheduling and employs the OPA algorithm to determine a feasible priority assignment. The schedulability analysis developed in this thesis builds directly upon this framework.

While most mixed-criticality scheduling research has focused on multiple WCET estimates across different criticality levels, other approaches have investigated adaptations of timing parameters. Tasks with different period specifications across criticality levels have been explored in [28]. When tasks are event triggered the period is an estimate and a higher criticality of a task may need to assume more frequent arrivals for certification. However, this work considers dynamic scheduling and does not address period adaptation in globally scheduled fixed-priority multiprocessor systems.

3.2 Security in real-time systems

A wide range of mechanisms have been proposed to enhance security in real-time systems. These approaches address security challenges from different perspectives, including modifying task execution behaviour, incorporating security considerations into scheduling decisions, and introducing hardware-assisted isolation and runtime monitoring mechanisms. Software-based techniques can often be deployed on existing platforms without requiring specialized hardware, making them broadly applicable to real-time systems. In contrast, hardware-assisted approaches, such as trusted execution environments (TEEs) and isolated memory regions, provide stronger isolation guarantees but require dedicated platform support.

The work in [5] introduces TaskShuffler, a software-based schedule randomization technique that mitigates timing side-channel attacks by reducing the predictability of real-time task execution. The approach is extended to globally scheduled fixed-priority multiprocessor systems in [8], demonstrating that schedule randomization can also be applied in multicore environments. While these approaches improve resistance against attacks that exploit deterministic execution patterns, [25] highlights limitations of relying solely on randomization. The authors show that schedule randomization does not necessarily prevent all schedule-based attacks and may, in some cases, introduce timing variations that increase an attacker’s success rate compared to traditional fixed-priority scheduling. Based on these observations, they argue for combining schedule-based techniques with additional security mechanisms, such as isolation-based approaches, to provide stronger protection against schedule-based attacks.

The approach in [6] introduces scheduling constraints aimed at preventing information leakage between tasks of different security levels. Unlike the randomization-based approach, this work modifies scheduler behavior to enforce isolation between security domains. The authors consider an adversary controlling one task and attempting to passively infer information from shared resources such as the cache. Given two tasks H and L , where H has a higher security level, a traditional FP scheduler may allow L to execute immediately after H . In such a case, L could inspect the shared cache and extract sensitive information. To avoid this, the following constraints are imposed:

- No instance of L may be scheduled immediately after any instance of H .
- If an instance of L is preempted by H , it cannot resume execution immediately after H completes.

To enforce these constraints, the authors propose two methods that rely on a flush task (FT) to clear the cache and eliminate residual information. The first method, Half-PreFlush (Half-PF), executes the FT whenever the scheduler transitions from a higher-security task to a lower-security one. The second method, ConstrainedPreFlush (CPF), reduces the number of flushes by preventing H from preempting L once L has begun executing.

They further examine how priority assignment can minimize the overhead introduced

by FT and preemption restrictions. When the priority ordering matches the security ordering (referred to as forward ordering), high-security tasks have higher priority and can frequently preempt lower-security tasks, resulting in more FT invocations and higher overhead. In contrast, backwards ordering where the ordering is reversed, high-security tasks rarely execute immediately before lower-security tasks, thereby reducing the number of flushes.

Their schedulability analysis shows that these constraints reduce the number of schedulable task sets compared to traditional FP scheduling. However, they also increase system security, allowing designers to trade real-time utilization for stronger isolation guarantees. The best results occur under backwards ordering, which yields schedulability close to traditional FP while still enforcing the security constraints. Moreover, at low system utilizations (up to approximately 50%), the constrained methods perform equivalently to FP, indicating that these techniques can be applied to many real-time systems without compromising schedulability [6].

In [29], the Multi-Phase Secure (MPS) model is proposed as a scheduling framework for systems that incorporate multiple security mechanisms, such as Trusted Execution Environments (TEEs), encryption co-processors, and FPGA-based secure computation. Different parts of a task may have distinct security requirements and therefore require different security mechanisms at different points during execution. To capture this, each task is divided into a sequence of phases, where each phase executes using a specific security mechanism. Transitions between phases incur overhead due to the startup and teardown of the corresponding security mechanism. Furthermore, if a task is preempted during a phase, this startup and teardown overhead is incurred on every preemption.

The work adopts the concept of Limited-Preemption Scheduling [30], in which each task τ_i is assigned a parameter β_i that specifies the maximum interval during which the task may execute without being preempted. Since every preemption within a phase incurs the startup and teardown overhead of the corresponding security mechanism, limiting preemptions reduces the overhead associated with security mechanism transitions. The authors present an algorithm for determining the schedulability of a task set by assigning appropriate values to β_i [29]. Their evaluation demonstrates improved schedulability compared to non-preemptive EDF scheduling. Overall, the work extends traditional real-time scheduling by explicitly accounting for the costs of switching between heterogeneous security mechanisms.

In [31], the Real-Time Mixed Trust Computing (RT-MTC) framework is introduced to enable trusted and untrusted components to coexist within the same CPS. The framework addresses the limitations of traditional isolation-based approaches, where untrusted functionality must be separated from trusted components and therefore cannot be directly utilized in safety-critical operations.

Within RT-MTC, a task may contain both untrusted functionality executed in a virtual machine and trusted functionality executed within a hypervisor. Logical enforcers monitor the outputs of untrusted components and replace unsafe results with safe alternatives, while temporal enforcers provide safe fallback behaviour when untrusted components fail to produce results within their required deadlines. To

support this execution model, RT-MTC introduces a scheduling approach that prioritizes trusted execution over untrusted execution, allowing safety mechanisms to intervene when necessary while maintaining real-time guarantees.

The works discussed so far primarily target uniprocessor systems and therefore do not address the additional challenges introduced by multiprocessor platforms, such as task migration, shared-resource contention, and increased scheduling complexity. These challenges require additional considerations when designing secure scheduling approaches for multiprocessor real-time systems.

The work in [9] introduces SecureCore, a framework designed to enhance the security of multicore real-time systems. The framework targets a dual-core processor architecture in which one core operates as the monitored core, while the other functions as a trusted secure core. The secure core is dedicated to system monitoring and intrusion detection and is capable of maintaining control of the physical system even if the monitored core is compromised.

To enable non-intrusive monitoring, SecureCore incorporates an on-chip hardware component known as the Timing Trace Module (TTM). The TTM continuously records execution-related information, including timestamps and program counter values, and stores this data in a scratchpad memory that is exclusively accessible by the secure core. Since the monitored core has no access to this memory, the monitoring process remains transparent and resistant to tampering by an attacker.

The primary intrusion detection mechanism, called GaIT (Gaussian Methods for Intrusion Detection Using Timing Profiles), exploits the deterministic timing characteristics of real-time tasks. The underlying assumption is that malicious code inevitably introduces additional execution time, thereby modifying the application's timing profile. Before deployment, GaIT constructs statistical models of the system's normal timing behaviour using Gaussian-based learning techniques. During operation, these models are used to distinguish genuine attacks from legitimate timing variations caused by factors such as shared-resource contention or other sources of execution-time variability commonly encountered in multicore systems.

During runtime, the secure core continuously compares the observed execution timing of each task against the learned timing profile. If the probability of the observed behaviour falls below a predefined threshold, the behaviour is classified as an intrusion. Upon detecting such an anomaly the secure core immediately transfers control to a safety controller, allowing the physical system to continue operating safely while the compromised core is reset and restored to a trusted state.

A limitation of SecureCore is that a full processor core is dedicated to monitoring and intrusion detection. In the evaluated dual-core architecture, this effectively reserves half of the available processing resources for security functionality, reducing the computational capacity available to application tasks.

The work in [32] presents HYDRA, a design-space exploration framework for integrating security mechanisms into multicore real-time systems. The framework is designed for legacy systems, where modifying the timing behaviour or execution parameters of the primary control tasks is not feasible without compromising system

correctness.

HYDRA adopts an opportunistic execution model in which security mechanisms are implemented as independent low-priority tasks that execute only when processing resources are available. By utilizing otherwise idle processor capacity, the framework enables security monitoring to be integrated without affecting the timing guarantees of the real-time application.

A key feature of HYDRA is the joint allocation of security tasks across processor cores and the selection of appropriate monitoring frequencies. This allows designers to balance the effectiveness of security monitoring against the available computational resources while preserving system schedulability. The authors evaluate the framework using a UAV control system case study and compare it with an approach that dedicates an entire processor core to security. Their results show that distributing security tasks across multiple cores improves intrusion detection performance while supporting higher overall system utilization.

Overall, HYDRA provides a practical approach for retrofitting security monitoring into multicore real-time systems, enabling additional security functionality without compromising the timing guarantees of the original application.

4

Task Model

The Mixed-Trust Task Model (MT) is considered on a homogeneous multiprocessor system with m processors. Similar to the mixed-criticality model, in which tasks are assigned different criticality levels and may execute with different parameters depending on the system's criticality level, the parameters of MT tasks depend on the current security level of the system.

In the MT model, each task is assigned one of two trust levels: *Trusted (TR)* or *Untrusted (UT)*. Untrusted tasks are permitted because they provide useful, non-safety-critical functionality. Although UT tasks are expected to satisfy their functional and timing requirements, they are not essential for maintaining the system's core functionality.

For example, a third-party library may satisfy a system's safety requirements while failing to meet its security certification. Although the library may provide its intended functionality correctly, it may also leak sensitive information to a third party or introduce other security vulnerabilities. Consequently, any task utilizing such a library would be classified as an untrusted task.

In this thesis, two security levels are considered: *Nominal Security (NS)* and *High Security (HS)*. The security level represents the current security posture of the system and determines the set of tasks that are eligible for execution. At the NS level, both Trusted (TR) and Untrusted (UT) tasks are permitted to execute. At the HS level, only TR tasks are permitted to execute, while UT tasks are suspended because they represent a potential attack vector.

A task set, denoted by Γ , consists of four types of tasks:

- Application tasks
- Security tasks
- Monitoring tasks
- High-Security tasks

Each task $\tau_i \in \Gamma$ is represented by the tuple

$$\tau_i = (L_i, C_i^{NS}, C_i^{HS}, T_i^{NS}, T_i^{HS}, D_i),$$

where

- $L_i \in UT, TR$ denotes the task's trust level.
- C_i^{NS} and C_i^{HS} denote the worst-case execution time (WCET) of the task at the NS and HS security levels, respectively.
- T_i^{NS} and T_i^{HS} denote the task period at the NS and HS security levels, respectively.
- D_i denotes the relative deadline, where $D_i \leq T_i^{NS}$ and $D_i \leq T_i^{HS}$.

The security-level-dependent parameters determine the execution behaviour of a task depending on the current security level. However, the parameters corresponding to a security level at which a task is inactive are undefined. Thus, C_i^{HS} and T_i^{HS} are undefined for UT tasks, while C_i^{NS} and T_i^{NS} are undefined for high-security tasks. Parameters for security and monitoring tasks, as well as trusted application tasks, are defined at both security levels.

A set of n application tasks, $\Gamma_{app} = \tau_1, \tau_2, \dots, \tau_n$, provides the core functionality of the system and may include both TR and UT tasks.

When the system transitions to the HS level, all UT tasks are suspended. This provides two important benefits. First, if an attacker has gained a foothold through an untrusted task, suspending that task may eliminate the attacker's access to the system. Second, suspending UT tasks frees processing capacity that can be reallocated to security-related activities.

Application tasks may also perform additional work when the system operates at the HS level. Thus, $C_i^{NS} \leq C_i^{HS}$. The additional execution time may be required to further isolate tasks, increasing context-switch overhead, or to insert execution-time padding that obfuscates power consumption and mitigates power side-channel attacks [33].

The set of security tasks, denoted by Γ_S , provides security services that are not implemented directly by the application tasks. This separation allows security mechanisms to be modelled independently from the application functionality. Consequently, the proposed model can support the integration of security mechanisms into legacy systems that were not originally designed with security functionality. These services may include cryptographic operations, such as encrypting communications or verifying message integrity, as well as system-level protection mechanisms. For example, a security task may flush shared hardware resources between task executions to remove residual information and prevent subsequent tasks from acquiring sensitive data through shared resources [29].

The characteristics of a security task may vary depending on the system security level. At the HS level, the task period may be reduced to allow more frequent execution, i.e., $T_i^{NS} \geq T_i^{HS}$, while the WCET may increase, $C_i^{NS} \leq C_i^{HS}$, to accommodate more computationally intensive security mechanisms, such as stronger encryption algorithms or more extensive integrity checks. It is assumed that the task deadline satisfies $D_i \leq T_i^{HS}$ ensuring that the deadline constraint remains valid at both security levels.

The set of monitoring tasks, denoted by Γ_M , is responsible for detecting attacks or abnormal system behaviour and initiating an appropriate response. For example, a monitoring task may perform file integrity checks to ensure that critical files have not been modified without authorization [34]. It may also monitor the execution behaviour of application tasks, since an attacker could cause a task to deviate from its expected execution pattern [9].

At the HS level, monitoring tasks may execute more frequently by reducing their periods, i.e., $T_i^{NS} \geq T_i^{HS}$. Again, it is assumed that the task deadline satisfies $D_i \leq T_i^{HS}$.

Finally, the high-security task set, Γ_{HS} , contains tasks that execute exclusively at the HS level and remain inactive during NS operation. Their purpose is to ensure that the system continues to function despite an attack. For example, these tasks may perform recovery actions following the detection of an attack that compromises system components [35]. Such actions may include restoring the system to its pre-anomaly state, enforcing additional security policies, or determining when the system can safely transition back to the NS level after a prescribed recovery period.

To illustrate how the Mixed-Trust Task Model operates, consider the task set shown in Table 4.1 executing on two processors. The set contains trusted and untrusted application tasks, a security task, a monitoring task, and a high-security task. A dash indicates that the corresponding parameter is undefined because the task is inactive at that security level.

Table 4.1: Example Mixed-Trust task set.

Task	Type	L_i	C_i^{NS}	C_i^{HS}	T_i^{NS}	T_i^{HS}	D_i
τ_1	Application	TR	5	6	10	10	10
τ_2	Application	UT	5	–	10	–	10
τ_3	Security	TR	3	4	10	10	10
τ_4	Monitoring	TR	3	3	10	8	8
τ_5	High-security	TR	–	3	–	20	20

During NS operation, τ_1 , τ_2 , τ_3 , and τ_4 are eligible for execution and execute periodically. The monitoring task τ_4 checks for security anomalies.

If τ_4 detects an attack, the system transitions to the HS level. The untrusted task τ_2 is then suspended and is no longer eligible for execution, while the high-security task τ_5 becomes active. The remaining tasks use their HS-level parameters, resulting in the active task set

$$\{\tau_1, \tau_3, \tau_4, \tau_5\}.$$

The utilization at the NS level is

$$U^{NS} = \frac{5}{10} + \frac{5}{10} + \frac{3}{10} + \frac{3}{10} = 1.6.$$

Since the system has two processors, this utilization is below the available processor capacity. This does not establish schedulability, but indicates that the task set is not ruled out by processor capacity alone. At the HS level, after suspending τ_2 , the utilization is

$$U^{HS} = \frac{6}{10} + \frac{4}{10} + \frac{3}{8} + \frac{3}{20} = 1.525,$$

which is also below the available processor capacity. In contrast, if the untrusted task τ_2 were allowed to continue executing after the transition, the HS-level utilization would become

$$U_{UT}^{HS} = 1.525 + \frac{5}{10} = 2.025 > 2.$$

Therefore, allowing τ_2 to continue executing would make the task set infeasible due to insufficient processor capacity, regardless of the scheduling algorithm.

This example illustrates one of the main benefit of the model: untrusted functionality can execute during nominal operation while computational resources can be reclaimed for security and recovery functionality when a security threat is detected. At the same time, trusted application tasks can continue to execute, allowing the system to maintain its core functionality during the security-level transition.

4.1 Scheduling algorithm

The dispatching mechanism of the MT scheduler operates as follows. A security level indicator l is maintained by the system and initialized to the Nominal Security level,

$$l \leftarrow NS.$$

At each scheduling instant, the scheduler determines the set of ready jobs that are eligible under the current security level l . The m highest-priority jobs from this set are selected and dispatched for execution on the m identical processors. If fewer than m eligible jobs are ready, all available eligible jobs are dispatched.

If a monitoring task detects an anomaly in the system, the security level is increased,

$$l \leftarrow HS.$$

Following a transition to the HS level, tasks that are not permitted under the new security level are suspended. Consequently, jobs belonging to UT tasks are no longer eligible for execution, while jobs belonging to high-security tasks in Γ_{HS} become eligible for execution.

The proposed dispatcher operates similarly to a traditional global fixed-priority scheduler on m processors, with the addition of a security-level indicator that determines task eligibility at runtime. This approach is inspired by the MSM algorithm

used for global mixed-criticality scheduling [10]. However, unlike MSM, where a criticality switch is triggered when a task exceeds its assumed WCET at the current criticality level, the MT scheduler transitions to the HS level when a monitoring task detects a security anomaly. Therefore, the mode transition is driven by the security state of the system rather than execution-time overrun, resulting in changes to task eligibility rather than changes based on criticality assumptions.

5

Schedulability analysis

The purpose of this analysis is to derive a schedulability test for the scheduling algorithm presented in Section 4.1. The primary challenge is accounting for the changes in task parameters that occur when the system undergoes a security-level transition between the NS and HS levels. In particular, the analysis must accommodate changes in task WCETs and periods, the suspension of untrusted tasks, and the activation of the high-security task set Γ_{HS} following a level transition.

The analysis considers two security levels: Nominal Security (NS) and High Security (HS). Let $\mathcal{L} \in \{NS, HS\}$ denote the current security level. The security level determines both the active task set and the corresponding task parameters.

At the NS level:

- Trusted and untrusted application tasks execute with WCET C_i^{NS} .
- Monitoring tasks execute with period T_i^{NS} .
- Security tasks execute with WCET C_i^{NS} and period T_i^{NS} .

At the HS level:

- Trusted application tasks execute with WCET C_i^{HS} .
- Untrusted application tasks are suspended.
- Monitoring tasks execute with period T_i^{HS} .
- Security tasks execute with WCET C_i^{HS} and period T_i^{HS} .
- The high-security task set Γ_{HS} is activated following the security-level transition.

The complete active task set is therefore defined as:

$$\Gamma = \begin{cases} \Gamma_{app} \cup \Gamma_M \cup \Gamma_S, & \text{if } \mathcal{L} = NS, \\ \Gamma_{app} \cup \Gamma_M \cup \Gamma_S \cup \Gamma_{HS}, & \text{if } \mathcal{L} = HS. \end{cases} \quad (5.1)$$

The response-time analysis (RTA) extends the analysis of multiprocessor mixed-criticality systems presented in [10] to the security-level model considered in this thesis. In the original mixed-criticality formulation, the system transitions between

LO- and HI-criticality levels. These criticality levels are mapped to the NS and HS security levels in the proposed model.

The transition from NS to HS requires additional execution time to be reserved for trusted tasks whose WCET increases after the transition. Based on the result in [10], if a task τ_j is schedulable before and after a security-level transition, its NS-level response time R_j^{NS} must satisfy

$$R_j^{NS} \leq \zeta_j \quad (5.2)$$

where

$$\zeta_j = \begin{cases} D_j - (C_j^{HS} - C_j^{NS}), & \text{if } L_j = TR, \\ D_j, & \text{if } L_j = UT. \end{cases} \quad (5.3)$$

The bound ζ_j accounts for the additional execution time that may be required after a transition to the HS level. A trusted task that has received C_j^{NS} units of execution before the transition may require an additional $C_j^{HS} - C_j^{NS}$ units of execution after the transition. Therefore, sufficient processor capacity must remain before the deadline to complete this additional execution. If this condition is not satisfied, the security-level transition may invalidate the schedulability guarantee and cause one or more tasks to miss their deadlines.

Consequently, the NS-level RTA uses ζ_j as the response-time bound of task τ_j . The HS-level RTA uses the original deadline D_j , since no further security-level transition requires additional execution time to be reserved.

The following sections present the RTA applied when the system remains at a fixed security level and the analysis of the transition to a higher security level.

5.1 Response-Time Analysis for a Fixed Security Level

This section presents the response-time analysis (RTA) for tasks executing within a fixed security level, i.e., when no security-level transition occurs. The analysis is applied independently at each security level using the corresponding task parameters.

At the NS security level, the analysis uses the NS-level task parameters C_i^{NS} and T_i^{NS} , and the response time is bounded by ζ_i . At the HS security level, the same analysis is applied using the HS-level task parameters C_i^{HS} and T_i^{HS} . In this case, the original task deadline D_i is used as the response-time bound. Thus, the response-time calculations remains unchanged, while the task parameters and deadline bound are determined by the current security level.

The response time $R_i^{\mathcal{L}}$ of a job of task τ_i is derived over a problem window of length t , where $\mathcal{L}$ denotes the security level under analysis. Within this problem window, the total interfering workload $I_i(t)$ is caused by higher-priority tasks $hp(i)$ executing

at the same security level. The system is schedulable at security level $\mathcal{L}$ if all tasks satisfy the corresponding response-time constraint. For the NS security level, this requires $R_i^{NS} \leq \zeta_i$, while for the HS security level this requires $R_i^{HS} \leq D_i$, where:

$$R_i^{\mathcal{L}} \leftarrow C_i^{\mathcal{L}} + \left\lfloor \frac{I_i(R_i^{\mathcal{L}})}{m} \right\rfloor \quad (5.4)$$

This is solved iteratively with a starting value $R_i^{\mathcal{L}} = C_i^{\mathcal{L}}$ on the right-hand side. The total interfering workload $I_i(t)$ consists of interference from higher-priority tasks $hp(i)$ and includes contributions from both carry-in (CI) and non-carry-in (NC) tasks. A carry-in task is a higher-priority task τ_k that was released before τ_i and is executing when τ_i is released.

For notational simplicity, the following equations use the deadline bound ζ_i . When the analysis is applied to the HS security level, ζ_i is replaced by the original deadline D_i .

The individual workload of a task τ_k in an interval of length t is given for NC and CI tasks respectively as:

$$W_k^{NC}(t) = \left\lfloor \frac{t}{T_k^{\mathcal{L}}} \right\rfloor \cdot C_k^{\mathcal{L}} + \min \left(C_k^{\mathcal{L}}, t - \left\lfloor \frac{t}{T_k^{\mathcal{L}}} \right\rfloor \cdot T_k^{\mathcal{L}} \right) \quad (5.5)$$

$$W_k^{CI}(t) = A_t^k \cdot C_k^{\mathcal{L}} + \min \left(C_k^{\mathcal{L}}, t + \zeta_k - C_k^{\mathcal{L}} - A_t^k \cdot T_k^{\mathcal{L}} \right) \quad (5.6)$$

where

$$A_t^k = \left\lfloor \frac{t + \zeta_k - C_k^{\mathcal{L}}}{T_k^{\mathcal{L}}} \right\rfloor. \quad (5.7)$$

The CI-workload is equivalent to the maximum workload of any sporadic task set with parameters (C,D,T) in an interval of length t [7]. The interfering workloads $I_{k,i}^{NC}(t)$ and $I_{k,i}^{CI}(t)$ of any task τ_k on τ_i are defined as follows:

$$I_{k,i}^{NC}(t) = \min \left(W_k^{NC}(t), \zeta_i - C_i^{\mathcal{L}} + 1 \right) \quad (5.8)$$

$$I_{k,i}^{CI}(t) = \min \left(W_k^{CI}(t), \zeta_i - C_i^{\mathcal{L}} + 1 \right) \quad (5.9)$$

The difference between the CI and NC interference of task τ_k is given by

$$I_{k,i}^{DIFF}(t) = I_{k,i}^{CI}(t) - I_{k,i}^{NC}(t) \quad (5.10)$$

The total interfering workload $I_i(t)$ from $hp(i)$ is calculated by the sum of all interfering workloads:

$$I_i(t) = \sum_{\tau_k \in hp(i)} I_{k,i}^{NC}(t) + \sum_{\tau_k \in \max(hp(i), m-1)} I_{k,i}^{DIFF}(t) \quad (5.11)$$

where $\max(hp(i), m-1)$ denotes the $(m-1)$ tasks in $hp(i)$ with the largest value of $I_{k,i}^{DIFF}(t)$, as there can be at most $m-1$ CI tasks [27].

5.2 Transitioning to a higher security level

If a monitoring task detects an anomaly, the security level of the system increases and tasks may change their execution parameters. Jobs of untrusted tasks are immediately suspended without allowing them to complete their execution, while other tasks may experience changes in WCET or period. As a result, the response time of a task executing during a security-level transition differs from the response time when the security level remains static. Let R_i^{trans} denote the response time of task τ_i during a security-level transition.

The response time R_i^{trans} is derived by considering an arbitrary job j_i^x of task τ_i , where x denotes the job index. The analysis considers the problem window $[r_i^x, r_i^x + t)$ of length t , where r_i^x is the release time of job j_i^x .

Let s denote the relative time instant, measured from r_i^x , at which the system transitions from the NS security level to the HS security level as shown in figure 5.1. At this instant, the task parameters are updated and the additional task set Γ_{HS} is activated. If $s > R_i^{NS}$, the job j_i^x completes before the transition occurs and is therefore unaffected by the security-level change. The analysis in this section therefore considers the case where the transition occurs while the job is executing, i.e., $0 \leq s < R_i^{NS}$.

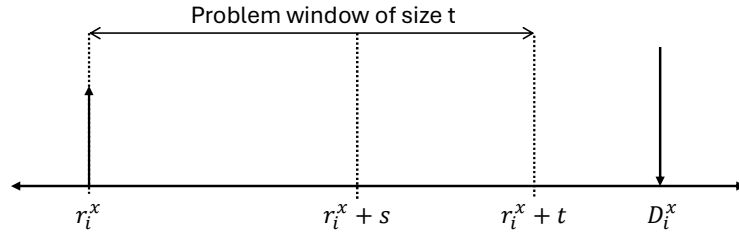


Figure 5.1: The problem window of size t over a level increase.

To determine the worst-case response time R_i^{trans} during a security-level transition, multiple transition points must be considered. Let $R_{i,s}^{trans}$ denote the response time of task τ_i when the transition occurs at time s . The response time $R_{i,s}^{trans}$ is calculated based on the workload and interference generated by each higher-priority task $\tau_k \in hp(i)$.

The workload contribution of a task depends on its execution across the security-level transition. The workload of an untrusted task is denoted by WU_k^{NC} and WU_k^{CI} for non-carry-in and carry-in cases, respectively. The workload of trusted tasks and security tasks is denoted by WT_k^{NC} and WT_k^{CI} for non-carry-in and carry-in cases, respectively.

The workload of a UT task τ_k with higher priority than τ_i , denoted as $hp_U(i)$, within the problem window $[r_i^x, r_i^x + t)$ is limited to the interval before the transition, $[r_i^x, r_i^x + s)$, since UT tasks are suspended at the transition instant s . Therefore, the NC and CI workload contributions of UT tasks are calculated using Equations 5.5 and 5.6 with $\mathcal{L} = NS$ as:

$$WU_k^{NC}(s, t) = W_k^{NC}(s) \quad (5.12)$$

$$WU_k^{CI}(s, t) = W_k^{CI}(s) \quad (5.13)$$

The upper bound on the NC workload of a TR task τ_k with higher priority than τ_i , i.e., $\tau_k \in hp_T(i)$, over the problem window $[r_i^x, r_i^x + t)$ is obtained by evaluating Equation 5.5 at the HS level. We denote this upper bound by $W_k^{UB}(t)$. Specifically, all jobs are assumed to execute for the HS-level execution time C_k^{HS} and to be released periodically with period T_k^{HS} , with the first job released at the beginning of the problem window.

For tasks with both HS and NS parameters, the problem window contains two distinct phases. Before the security-level transition at s , the task executes with NS-level parameters, contributing $\lfloor s/T_k^{NS} \rfloor$ jobs of at most C_k^{NS} each. The last job of τ_k released before s , denoted j_k^* , is released at

$$r_k^* = \left\lfloor \frac{s}{T_k^{NS}} \right\rfloor \cdot T_k^{NS}.$$

If this job's deadline has not expired at the transition instant, i.e., $r_k^* + \zeta_k > s$, the job may not have completed and its remaining execution must be considered after the transition as shown in figure 5.2. In this case, W_k^{UB} is applied over the interval $[r_k^*, t)$.

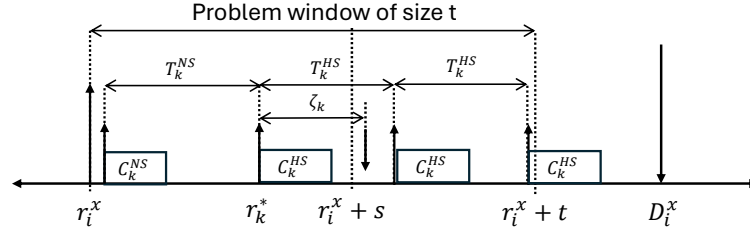


Figure 5.2: The job j_k^* contributes C_k^{HS} to the NC workload as the transition happens before $r_k^* + \zeta_k$.

Conversely, if $r_k^* + \zeta_k \leq s$, the boundary job completes using NS-level parameters before the transition and does not contribute additional execution with HS-level parameters. The next job considered with HS-level parameters is released at $r_k^* + T_k^{HS}$, and the corresponding HS-level workload is added to the post-transition contribution as shown in figure 5.3.

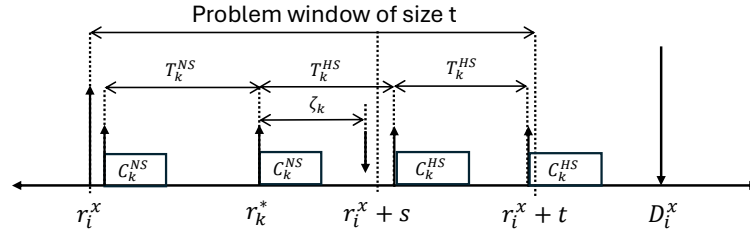


Figure 5.3: The job j_k^* contributes C_k^{NS} to the NC workload as the transition happens after $r_k^* + \zeta_k$.

Tasks $\tau_k \in \Gamma_{HS}$ are a special case, as they have no NS-level parameters and are not active before the security-level transition. Their workload is only relevant over the interval $[r_i^x + s, r_i^x + t)$, which has the length $t - s$ as shown in figure 5.4.

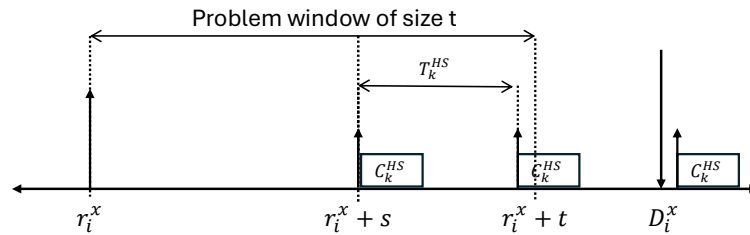


Figure 5.4: The first release of the high security task τ_k appear at the point s when the level increases.

The NC workload of a trusted task $WT_k^{NC}(s, t)$ is therefore:

$$WT_k^{NC}(s, t) = \begin{cases} W_k^{UB}(t - s) & \text{if } \tau_k \in \Gamma_{HS} \\ W_k^{UB}(t - r_k^*) + \left\lfloor \frac{s}{T_k^{NS}} \right\rfloor \cdot C_k^{NS} & \text{if } r_k^* + \zeta_k > s \\ W_k^{UB}(t - r_k^* - T_k^{HS}) + \left(\left\lfloor \frac{s}{T_k^{NS}} \right\rfloor + 1 \right) \cdot C_k^{NS} & \text{if } r_k^* + \zeta_k \leq s \end{cases} \quad (5.14)$$

The CI workload of a TR task is considered for a pattern of release that maximises the amount of workload in the problem window.

- A job of task τ_k is released at the end of the problem window ($r_i^x + t - C_k^{TR}$). Any job released after this executes as early as possible.
- Any job released before ($r_i^x + t - C_k^{TR}$) executes as late as possible.

Figure 5.5 shows the release pattern in a problem window of size t . Each job released before the transition job is executing for C_k^{NS} as close to the deadline as possible.

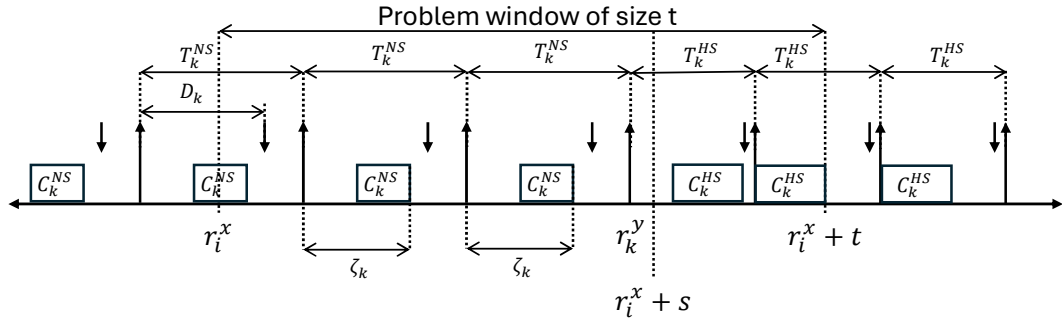


Figure 5.5: Release pattern to maximise workload of task τ_k in the problem window of size t when a level increase occurs at time $r_i^x + s$.

With this release pattern the CI workload $WH_k^{CI}(s, t)$ can be calculated by first calculating the workload of τ_k and then to shift the problem window left or right to determine the maximum increase in workload due to any shift of the problem window. A job j_k^y splits the problem window into two such that any job released before j_k^y executes for C_k^{NS} with period T_k^{NS} while j_k^y and any job released after executes with HS -parameters. Since j_k^y is executing during the transition the analysis assumes it is executing using HS -parameters from its release to account for an eventual increase in the WCET and to account for an earlier release of the next job if the frequency is increased. The release time r_k^y of this job can be precisely determined using:

$$r_k^y = (r_i^x + t) - C_k^{HS} - \mathcal{N}_{t,s}^k \cdot T_k^{HS} \quad (5.15)$$

Where $\mathcal{N}_{t,s}^k$ is the number of jobs released with HS parameters in the window $[r_k^y, r_i^x + t - C_k^{HS})$:

$$\mathcal{N}_{t,s}^k = \left\lceil \frac{\max(0, t - C_k^{HS} - s)}{T_k^{HS}} \right\rceil \quad (5.16)$$

τ_k is a trusted task which makes the upper bound of the response time of any job released before j_k^y , $\zeta_k = D_k - (C_k^{HS} - C_k^{NS})$ according to equation 5.3. This means the jobs released earlier completes their execution at least $D_k - \zeta_k = (C_k^{HS} - C_k^{NS})$ time units before their deadlines. The execution of j_k^{y-1} finishes at or before $r_k^y - (T_k^{NS} - \zeta_k)$ and therefore the workload of τ_k within $[r_i^x, r_k^y)$ can be refined into the workload within $[r_i^x, r_k^y - (T_k^{NS} - \zeta_k))$. Let Q be the length of $[r_i^x, r_k^y - (T_k^{NS} - \zeta_k))$ such that:

$$Q = \max\{0, t - C_k^{HS} - \mathcal{N}_{t,s}^k \cdot T_k^{HS} - (T_k^{NS} - \zeta_k)\} \quad (5.17)$$

If $Q = 0$ it means that j_k^y is the first job of τ_k in the problem window which in turn means that all jobs of τ_k executes with TR-parameters in the full interval. The CI-workload in this case can be calculated by using equation 5.6 with $\mathcal{L} = HS$ and replacing ζ_k with D_k .

If $Q > 0$ the combined workload of jobs before and after r_k^y need to be considered. We denote this workload $\mathcal{PQ}_k(s, t)$. The workload of jobs executing before have a workload that can be calculated with equation 5.5 with Q as the interval and task parameters from the NS mode. At most $\mathcal{N}_{t,s}^k + 1$ jobs execute after r_k^y . Within the entire window the workload $\mathcal{PQ}_k(s, t)$ is then:

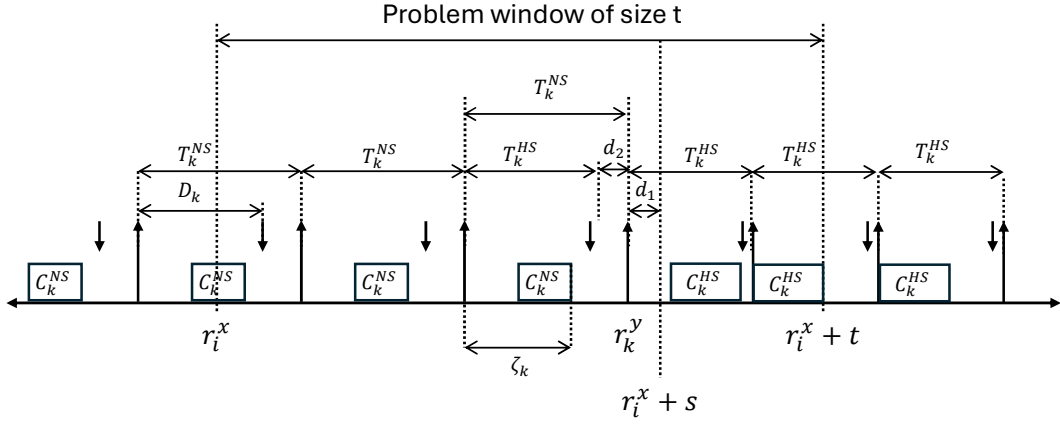
$$\mathcal{PQ}_k(s, t) = W_k^{NC}(Q) + (\mathcal{N}_{t,s}^k + 1) \cdot C_k^{HS} \quad (5.18)$$

[10] showed that the net increase in workload due to any shift of the problem window is bounded by $C_k^{HS} - C_k^{NS}$ when there is no period difference between different level, e.g. $T_k^{HS} = T_k^{NS}$. However, when the period becomes shorter after the level increase, the worst case is slightly different, as the release r_k^y may occur earlier if s occur earlier.

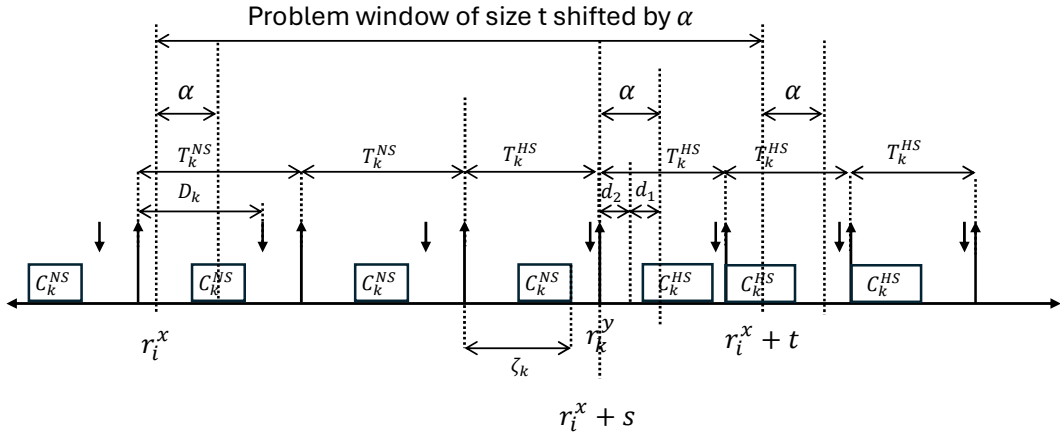
A shift of a full period T_k^{NS} results in exactly the same pattern as no shift. Therefore, it is sufficient to consider shifts in the range from 0 to T_k^{NS} . Moreover, shifting the problem window to the left by an amount α is equivalent to a pattern that occurs when shifting it to the right by $T_k^{NS} - \alpha$. Thus, it is sufficient to consider shifts in only one direction.

Consider shifting the problem window to the left by an amount α , where $0 \leq \alpha \leq T_k^{NS}$. As α increases, at most C_k^{NS} may enter from the left-hand side of the problem window in one period T_k^{NS} . Additionally, the job $j_k^{(y-1)}$ may experience the level increase and increase its execution time by $C_k^{HS} - C_k^{NS}$ due to an earlier level increase. Combining these gives a maximum upper bound on the increase of $C_k^{NS} + (C_k^{HS} - C_k^{NS}) = C_k^{HS}$. However, work may also exit through the right-hand side of the problem window when shifting it to the left. Let d_1 denote the distance between $r_i^x + s$ and r_k^y , additionally let d_2 denote the distance $T_k^{NS} - T_k^{HS}$ as seen in figure 5.6.

For a leftward shift in the range $0 \leq \alpha \leq d_1$, s happens after the release of j_k^y and the workload decreases on the right-hand side by $\min(\alpha, C_k^{HS})$. The increase


 Figure 5.6: Problem window of size t before a shift

in workload in this range is at most $\min(\alpha, C_k^{NS})$. Thus, during this range, the decrease in workload compensates for the corresponding increase that may enter from the left-hand side.


 Figure 5.7: Problem window of size t shifted to the left by $\alpha = d_1 + d_2$.

For a leftward shift in the range $d_1 \leq \alpha \leq d_2$, s is pushed earlier than the original release of r_k^y . This causes r_k^y to be pushed earlier as well. It can be moved as early as T_k^{HS} from $r_k^{(y-1)}$. In the leftward shift $d_1 \leq \alpha \leq d_2$, at most d_1 workload can leave from the right-hand side, as all releases are pushed earlier together with r_k^y by α . During this shift workload can accumulate without any workload leaving from the right-hand side. In the worst case $d_1 = 0$ which means that the net increase may be as large as $\min(C_k^{NS}, d_2) = \min(C_k^{NS}, (T_k^{NS} - T_k^{HS}))$. In the case when $d_1 > 0$ some workload have been pushed out of the problem window reducing the net increase. The upper bound on the increase in this range is therefore C_k^{NS} .

Figure 5.7 shows how a shift of the problem window of figure 5.6 by $\alpha = d_1 + d_2$ causes the release r_k^y to occur earlier, pushing all consecutive releases earlier as well.

Some workload has entered on the left hand side and an amount d_1 workload has been reduced on the right hand side.

For a leftward shift in the range $d_1 + d_2 \leq \alpha \leq T_k^{NS}$, workload leaves the right-hand side at a rate of α to at least C_k^{HS} . The total workload decrease is at least $\min(C_k^{HS}, \alpha - d_2)$. The job $j_k^{(y-1)}$ increases its execution time when the shift is large enough such that s occur before the finished execution of $j_k^{(y-1)}$ at ζ_k . The additional execution time is $D_k - \zeta_k = C_k^{HS} - C_k^{NS}$. A further shift of $C_k^{HS} - C_k^{NS}$ is needed to move s past this additional execution, during which at least the same amount of workload leaves, or has already left, from the right-hand side. To reach this point $\alpha = d_1 + d_2 + C_k^{HS} - C_k^{NS}$ and the workload decrease is $\min(C_k^{HS}, \alpha - d_2) = \min(C_k^{HS}, d_1 + C_k^{HS} - C_k^{NS})$, worst case being $d_1 = 0$ as mentioned previously. Thus, the additional execution of $j_k^{(y-1)}$ is accompanied by at least the same amount of decrease in workload from the right-hand side.

During part of this range, work may increase at a higher rate than it leaves from the right, particularly when the increase in execution time of $j_k^{(y-1)}$ overlaps with new workload entering the problem window from the left. However, this does not increase the worst-case net increase beyond C_k^{NS} . Any workload entering during this range is accompanied by workload leaving from the right, and the cumulative increase can therefore only be as large as the maximum increase from the left hand side, C_k^{NS} .

Therefore, the maximum net increase from any shift of the problem window is C_k^{NS} when both the execution time and period are changed after an increase in security level.

In the case when only the period changes, that is, $C_k^{HS} = C_k^{NS}$, the only increase can come from the left-hand side of the problem window and is at most C_k^{NS} . This increase is cancelled out by the job finishing execution right before the end of the problem window and being shifted out, except when the shift moves the release r_k^y earlier, e.g. $d_1 \leq \alpha \leq d_2$. The maximum increase in this case is $\min(C_k^{NS}, (T_k^{NS} - T_k^{HS}))$.

The carry in workload of a trusted task τ_k in the problem window is given as follows:

$$WT_k^{CI}(s, t) = \begin{cases} \mathcal{P}Q_k(s, t) + \beta & \text{if } Q > 0 \\ \mathcal{W}_k^{CI}(t) & \text{Otherwise} \end{cases} \quad (5.19)$$

where

$$\beta = \begin{cases} C_k^{HS} - C_k^{NS} & \text{if } T_k^{NS} = T_k^{HS} \\ \min(C_k^{NS}, T_k^{NS} - T_k^{HS}) & \text{if } C_k^{NS} = C_k^{HS} \\ C_k^{NS} & \text{otherwise} \end{cases} \quad (5.20)$$

The interfering workload of a task τ_k upon τ_i , denoted $\bar{I}_{k,i}^{NC}(s, t)$ and $\bar{I}_{k,i}^{CI}(s, t)$ are calculated as follows:

$$\bar{I}_{k,i}^{NC}(s, t) = \begin{cases} \min\{WU_{k,i}^{NC}(s, t), D_i - C_i^{HS} + 1\} & \text{if } \tau_k \in hpU(i) \\ \min\{WT_{k,i}^{NC}(s, t), D_i - C_i^{HS} + 1\} & \text{if } \tau_k \in hpT(i) \end{cases} \quad (5.21)$$

$$\bar{I}_{k,i}^{CI}(s, t) = \begin{cases} \min\{WU_{k,i}^{CI}(s, t), D_i - C_i^{HS} + 1\} & \text{if } \tau_k \in hpU(i) \\ \min\{WT_{k,i}^{CI}(s, t), D_i - C_i^{HS} + 1\} & \text{if } \tau_k \in hpT(i) \end{cases} \quad (5.22)$$

The difference between the CI and the NC workload of τ_k is given by:

$$\bar{I}_{k,i}^{DIFF}(s, t) = \bar{I}_{k,i}^{CI}(s, t) - \bar{I}_{k,i}^{NC}(s, t) \quad (5.23)$$

Now the upper bound of the total interfering workload $\bar{I}_i(s, t)$ due to all $hp(i)$ within the problem window of length t can be calculated:

$$\bar{I}_i(s, t) = \sum_{\tau_k \in hp(i)} \bar{I}_{k,i}^{NC}(s, t) + \sum_{\tau_k \in \max(hp(i), m-1)} \bar{I}_{k,i}^{DIFF}(s, t) \quad (5.24)$$

Where $\max(hp(i), m-1)$ is the $m-1$ tasks from $hp(i)$ with the largest value of $\bar{I}_{k,i}^{DIFF}(s, t)$. The interference is an integer and all m processors are busy when tasks from $hp(i)$ are executing an upper bound on the interference on a task τ_i within the problem window of length t is given as $\lfloor \bar{I}_i(s, t)/m \rfloor$.

Finally the response time test of a TR-task τ_i can be iteratively computed with the following equation starting with $R_{i,s}^{trans} = C_i^{HS}$ on the RHS:

$$R_{i,s}^{trans} \leftarrow C_i^{HS} + \left\lfloor \frac{\bar{I}_i(s, R_{i,s}^{trans})}{m} \right\rfloor \quad (5.25)$$

The response time is the largest value given in the range $0 \leq s \leq R_i^{NS}$.

5.3 Priority assignment and schedulability

Section 2.1.3 listed three conditions for applying OPA to multiprocessor systems. The analysis in this chapter depends only on independent properties of higher-priority tasks, namely their WCET, period, and deadline, thereby satisfying Condition C1. Because the analysis does not rely on any properties of lower-priority tasks, Condition C2 is fulfilled. Finally, the analysis considers interference only from higher-priority tasks; therefore, if two adjacent tasks swap priorities, the task that receives the higher priority cannot experience more interference than before, satisfying Condition C3. Consequently, the analysis presented here is compatible with OPA for multiprocessor systems.

Using the OPA algorithm described in Section 2.1.3, the response times for NS, HS, and the transition can be used to determine whether a task can be assigned a given priority. The following schedulability test checks whether an individual task satisfies the response-time requirements for a given priority.

```

1  SchedulabilityTest(taskset, task, priority)
2
3  R_NS <- compute using equation 5.4 with L = NS
4  R_HS <- compute using equation 5.4 with L = HS
5  R_trans <- compute using equation 5.25 for 0 <= s <= R_NS
    
```

```
6
7  if all response times meet their deadlines:
8      return Yes
9  else:
10     return No
```

The procedure returns *Yes* if the task satisfies all response-time requirements and can therefore be assigned the considered priority. If the procedure returns *No*, the task is not schedulable with the considered priority. Combined with OPA, this test can be used to determine whether a feasible priority assignment exists for the complete task set.

6

TaskShufflerGlobal: A Randomization Scheduling Algorithm

This chapter presents a new extension of TaskShuffler [5], referred to as TaskShufflerGlobal, for globally scheduled fixed-priority multiprocessor systems. The extension provides schedule randomization to mitigate timing-based attacks. In the MT-model, the focus is on preventing an attacker from gaining knowledge during the NS-level. During the HS-level, only trusted tasks are executing making randomization unnecessary.

As described in Section 2.3, the maximum inversion budget V_i must be calculated for each task prior to runtime. For the m highest-priority tasks on m processors, the inversion budget simplifies to $V_i = \zeta_i - C_i^{NS}$. These tasks experience no interference from other tasks during normal execution, as there are m processors available.

For lower-priority tasks, interference from higher-priority tasks must be considered when determining the inversion budget. The maximum interference that can occur at security level NS is used for this purpose. By evaluating the maximum interference up to ζ_i using equation 5.11, the interference that a job may experience before reaching ζ_i can be determined. The resulting inversion budget is given by:

$$V_i = \zeta_i - \left(C_i^{NS} + \left\lfloor \frac{I_i(\zeta_i)}{m} \right\rfloor \right) \quad (6.1)$$

Each job released by task τ_i inherits the inversion budget V_i of its corresponding task. A non-positive inversion budget indicates that a job has no remaining capacity for priority inversion and therefore cannot tolerate further delay without potentially missing its deadline.

At each scheduling decision, TaskShufflerGlobal is provided with the ready queue sorted in priority order. The algorithm maintains two lists: EX , containing the jobs selected for execution following the scheduling decision, and LC , containing candidate jobs that can be selected to fill EX . The ready queue is traversed in priority order.

The EX list is first populated with the m highest-priority jobs in the ready queue

that have $V_i \leq 0$. These jobs are forced to execute in the current scheduling decision because they have no remaining inversion budget, and any further preemption could cause them to miss their deadlines. The remaining jobs among the m highest-priority jobs are added to LC as candidates for execution.

While there remains processor capacity, jobs are added to LC from the ready queue until a job with $V_i \leq 0$ is encountered. This job defines the boundary of the candidate set, as it cannot be further preempted without risking a deadline miss. We denote this job by M^B , representing the lowest-priority job that can currently be considered for random selection. More precisely, M^B is the highest-priority job with a non-positive inversion budget among the jobs below the m highest-priority jobs that has not yet been selected for execution. M^B is related to the minimum inversion priority M_i described in Section 2.3; however, while M_i is computed offline for each task, M^B is determined dynamically during each scheduling decision based on the current state of the ready queue and the currently selected jobs for execution. Adding any jobs with lower priority than M^B would allow M^B to be preempted, potentially causing a deadline miss. A selection process is started, during which jobs are randomly selected from LC and added to EX . If the selected job is M^B , the selection process terminates. Since M^B will execute, it cannot be preempted during the current scheduling decision, and the ready queue can then be traversed further to add additional jobs to LC . The next job with a non-positive inversion budget encountered during the traversal is designated as the new M^B and triggers the next selection process.

This process is repeated until m jobs have been selected for execution or the entire ready queue has been traversed. If the entire ready queue is traversed while EX still has available capacity, all jobs in LC have positive inversion budgets, or there were fewer than m jobs in the ready queue. In this case, processors may remain idle instead of selecting another job. The idle option is treated as having the lowest priority. Allowing processors to idle increases the number of possible scheduling decisions and may increase the scheduling entropy compared to always executing jobs when available [5].

The TaskShufflerGlobal algorithm for a fixed-priority scheduler with m processors is provided below.

```

1 TaskShufflerGlobal(readyQ)
2 LC = []
3 EX = []
4
5 for i = 0 to |readyQ|:
6     job = readyQ[i]
7
8     // add the m highest priority jobs to LC or EX
9     if i < m:
10         if job.V <= 0:
11             EX.append(job)
12         else:
```

```

13         LC.append(job)
14
15     // Append lower-priority jobs to LC
16     // until a job with  $V \leq 0$  is encountered
17     else:
18         LC.append(job)
19         if job.V  $\leq 0$ :
20             MB = job
21             // Randomly pick jobs from LC and move to EX
22             // until |EX| is equal to m or MB is picked
23             while |EX| < m:
24                 ex_job = pick_and_remove_random(LC)
25                 EX.append(ex_job)
26                 if ex_job == MB:
27                     break
28
29         if |EX| == m:
30             break
31
32     // If all members of the readyQ have been added to LC
33     // pick a job from LC or with equal probability, idle
34     while |EX| < m and |LC| > 0:
35         with idle probability  $1/(|LC|+1)$ 
36         EX.append(idle)
37     else:
38         job = pick_and_remove_random(LC)
39         EX.append(job)
40
41     return EX

```

The for loop starting on line 5 traverses the ready queue in priority order and builds the set of jobs eligible for random selection. The loop first handles the m highest-priority jobs and then adds lower-priority jobs until a job with a non-positive inversion budget is encountered.

Lines 9–13 handle the m highest-priority jobs in the ready queue. For each job, its inversion budget is checked. If $V_i \leq 0$, the job is immediately added to EX , since it must execute to avoid a potential deadline miss. Otherwise, the job is added to LC and may be selected for execution through randomization.

Once the m highest-priority jobs have been considered, lower-priority jobs are handled in lines 17–27. Jobs are added to LC one at a time. When a job with $V_i \leq 0$ is encountered on line 19 it is designated as M^B and a procedure to pick jobs from LC is triggered. On lines 24–25, a job is randomly picked from LC and moved to EX . If the picked job is M^B , the procedure terminates on lines 26–27, and the outer loop continues to traverse the ready queue. Since the boundary job has now been selected for execution, it cannot be preempted during the current scheduling decision. If instead the procedure fills EX to capacity, the outer loop is terminated

on lines 29–30.

The final while loop starting on line 34 handles the case when all jobs from the ready queue have been traversed without filling EX to m jobs. If this point is reached, all jobs in LC have a positive inversion budget, or LC is empty. If a non-positive-budget job had been encountered before the end of the ready queue, the inner procedure would have been triggered on line 19 and would have continued until either that job was selected or EX reached capacity. Line 35 determines whether a processor will idle or execute a job. At each selection, the idle option and each remaining job in LC are equally likely to be selected. With 4 jobs in LC and one idle option, each option has a probability of $\frac{1}{4+1}$. Selected jobs are removed from LC , while the idle option remains available for subsequent selections. Thus, more than one processor may be idle after the same scheduling decision. Finally, EX is returned containing the jobs selected for execution.

At each scheduler tick, the inversion budgets of priority-inverted jobs are updated. If the inversion budget of a job is exhausted, a new scheduling decision is triggered to ensure that the job is not delayed beyond the point at which it can meet its deadline. The budget update procedure is given below.

```

1 DecreaseInversionBudget(readyQ, EX):
2
3 for job in readyQ:
4     if job not in EX:
5         if job.priority > lowest priority in EX:
6             job.V--;
7         if (job.V == 0):
8             trigger new scheduling decision

```

The procedure decrements the inversion budget of each ready job that has a lower priority than the lowest-priority job currently executing. Such a job is priority-inverted and therefore consumes one unit of its inversion budget. When the budget reaches zero, a new scheduling decision is triggered, ensuring that the job is selected for execution before further inversion can cause a deadline miss.

An example using $m = 3$:

A ready queue sorted in priority order is shown in Table 6.1.

Job	J_1	J_2	J_3	J_4	J_5	J_6	J_7
V_i	0	4	3	0	2	1	3

Table 6.1: Jobs in the ready queue with their corresponding inversion budgets.

Running TaskShufflerGlobal on the ready queue could proceed as follows. The m highest-priority jobs are J_1 , J_2 , and J_3 . Of these, J_1 has no inversion budget available and is therefore added to EX , while J_2 and J_3 are added to LC . At this point, $EX = [J_1]$ and $LC = [J_2, J_3]$.

The next job in the ready queue is J_4 , which has $V_i = 0$. It is therefore added to LC and designated as the boundary job M^B . The selection process is then triggered

with $LC = [J_2, J_3, J_4]$. A job is randomly selected from LC and moved to EX . In this example, J_4 is selected. Since $J_4 = M^B$, the selection process terminates. The current state is therefore $EX = [J_1, J_4]$, and $LC = [J_2, J_3]$.

The traversal then continues with the remaining jobs in the ready queue. Since no further job with a non-positive inversion budget is encountered, the remaining jobs can be added to LC . There is now one remaining position in EX . The five remaining jobs in LC and the idle option are equally likely to be selected, giving each option a probability of $\frac{1}{5+1}$. In this example, the idle option is selected. The final result of the scheduling decision is therefore $EX = [J_1, J_4, idle]$.

The remaining jobs are not selected for execution and therefore remain ready for the next scheduling decision. Since one processor is idle, the idle option is treated as having the lowest priority. All jobs that are not executing will decrease their inversion budget by one for each subsequent system tick. In this example, J_6 has the smallest positive inversion budget among the non-executing jobs, with $V_i = 1$. Its inversion budget therefore decreases to 0 first, triggering a new scheduling decision.

7

Experimental Setup

This chapter describes the setup used to evaluate the two contributions of this thesis. The first part test the effectiveness of the proposed schedulability test for the mixed-trust model. The second part describes the simulation setup and runtime metrics used to evaluate the randomization algorithm.

7.1 Task generation

Task sets were generated using the UUnifast-Discard algorithm [36], a multiprocessor extension of the UUnifast algorithm. UUnifast generates a uniform distribution of n task utilizations whose sum is equal to a specified target utilization, U_t . In multiprocessor systems, U_t can exceed 1, in which case UUnifast can produce individual utilizations exceeding 1. UUnifast-Discard addresses this by rejecting any set containing a utilization greater than 1 and restarting the generation process. The pseudocode for UUnifast-Discard is provided below.

```
1  UUnifastDiscard(n, Ut, discard_limit):
2      sumU = Ut
3      U = []
4      attempts = 0
5
6      while |U| < n:
7          if attempts >= discard_limit:
8              return failure
9
10         nextSumU = sumU * pow(rand(), 1/(n-1))
11         Ui = sumU - nextSumU
12
13         if Ui > 1.0:
14             U = []                // discard and restart
15             sumU = Ut
16             attempts++
17         else:
18             U.append(Ui)
19             sumU = nextSumU
20
21     return U
```

The algorithm takes n , U_t , and a discard limit as inputs. Starting with U_t as the remaining utilization, it iteratively generates utilization values by randomly splitting the remaining amount (lines 10–11). If a generated value exceeds 1, the current set is discarded and the process restarts. Otherwise, generation continues with the remaining utilization. The process terminates when n valid utilizations have been generated or the discard limit is reached.

A weakness of the UUnifast-Discard algorithm is the sharply increasing discard rate when $U_t > n/2$ [36]. For example, with $U_t = 10.0$ and $n = 16$, only 2.6% of generation attempts succeed. This can be addressed by halving both quantities to $U_t = 5.0$ and $n = 8$, which restores a near-perfect success rate. Two such independent generations can then be combined to reach the original target utilization. However, as $U_t \rightarrow n$ the space of feasible task sets shrinks, eventually making even this approach impractical. The impact of this limitation is mitigated by choosing an appropriate number of tasks per generation.

If the algorithm succeeds, a set of n utilization values is returned. The utilization of a task is computed as $U_i = \frac{C_i}{T_i}$. Thus, a task with a given utilization can be constructed by randomly choosing a period and multiplying it by the utilization to determine the execution time. To keep the task-set hyperperiod manageable during runtime simulation, periods were constrained to powers of two within the range $[2^3, 2^{10}]$ ms, as well as their multiples by 3 and 5, up to a maximum of 1024 ms. This range approximately covers the $[10, 1000]$ ms interval typical for real-time systems. The maximum hyperperiod resulting from these period values is $1024 \cdot 3 \cdot 5 = 15360$ ms.

The UUnifast-Discard algorithm produces continuous utilization values in the range $[0.0, 1.0]$. However, task parameters must be represented as integers, making rounding unavoidable when computing execution times. Consequently, the total utilization of the resulting task set may differ from U_t . A tolerance of 0.005 was used to keep the resulting task-set utilization close to the target. The discretization also reduces the uniformity of the generated utilizations, as multiple continuous utilization values may map to the same integer-valued task parameters. Nevertheless, UUnifast-Discard provides a broadly uniform spread of utilization values before discretization, making it suitable for generating diverse task sets across the desired utilization range.

Since the MT model can execute at two levels with different task parameters, the task utilization differs between the *NS* and *HS* levels. The *NS* level was used as the baseline, with the target utilization applied to application, monitoring, and security tasks executing at this level. The generated utilization values were used to derive their *NS*-level parameters, T_i^{NS} and C_i^{NS} . Since tasks in Γ_{HS} do not execute at the *NS* level, they were generated separately using a target utilization U_{HS} . To evaluate the impact of introducing these tasks after a level increase, U_{HS} was varied over the range $[0, 1.5]$.

After utilization has been generated an integer period, T_i^{NS} , from the available periods was chosen randomly. This was then used to calculate a WCET by multiplying the utilization with the period, rounded down to get an integer, $C_i^{NS} = \lfloor T_i^{NS} \cdot U_i \rfloor$. For small U_i the WCET often becomes 0, in this case the WCET was set to 1 to

avoid excessive amounts of discards until one fits. Due to rounding down the final task was adjusted to try to keep the utilization close to the target utilization. If the difference of the actual utilization after integer conversion was larger than the tolerance 0.005 it was discarded and a new attempt was made. The deadlines of untrusted tasks was set equal to the period $D_i = T_i^{NS}$.

The number of trusted tasks was varied from 0% to 100% of the application tasks. The number of application tasks was varied across 10, 15, and 20. The numbers of monitoring, security, and high-security tasks were fixed at 3. Three tasks per category were considered sufficient to represent each task type with multiple instances while maintaining a consistent task-set structure. In the smallest configuration, with 10 application tasks, the nine monitoring, security, and high-security tasks already account for 47% of the total tasks. Increasing their number would therefore substantially change the composition of the task sets, while the number of these task types is not a parameter under investigation.

For trusted application tasks and security tasks a parameter C_{scale} in the range $[1.0, 1.5]$ was used to determine $C_i^{HS} = C_{scale} \cdot C_i^{NS}$. For monitoring tasks, a parameter $T_{scale} = 0.75$ was used to scale the period according to $T_i^{HS} = T_{scale} \cdot T_i^{NS}$. Since the periods were chosen from a fixed set of possible values, the closest available period was selected after scaling. For example, if $T_i^{NS} = 192$, scaling gives $192 \cdot 0.75 = 144$. Since 144 is not in the set of possible periods, the closest available value that does not reduce the period beyond the intended scaling factor of 0.75, 160, was selected. The actual scaling factor was therefore $\frac{160}{192} = 0.833$. A greater reduction in the period would result in a substantial increase in utilization at the HS level, making most task sets infeasible. Due to the limited number of available periods, finer-grained scaling factors are not possible with the chosen period set. The deadlines of tasks with varying periods were set to their HS -level periods, $D_i = T_i^{HS}$.

If any task utilization due to increased execution time or decreased period in the HS parameters resulted in $U_i > 1$, a new task set was generated.

7.2 Schedulability Test

With task sets generated according to Section 7.1, each task set was evaluated for schedulability using the schedulability test described in Section 5.3 with the OPA algorithm. For each target utilization, 1000 different task sets were generated and evaluated. The *acceptance ratio* is defined as the fraction of generated task sets that are classified as schedulable. Let $AR(U)$ denote the acceptance ratio at target utilization U :

$$AR(U) = \frac{\text{Number of accepted task sets}}{\text{Number of generated task sets}}. \quad (7.1)$$

For example, if 678 out of 1000 generated task sets are classified as schedulable, the acceptance ratio is $AR(U) = 0.678$.

In addition to the acceptance ratio, a *weighted acceptance ratio* can be used to obtain a single aggregate measure of the acceptance ratios across a range of utilization values [37]. This is useful when investigating the effect of another parameter, such as the percentage of trusted tasks. For each value of the parameter being investigated, the weighted acceptance ratio is calculated over the considered utilization values. Let S denote the set of utilization values and let $W(S)$ denote the corresponding weighted acceptance ratio. It is calculated as follows:

$$W(S) = \frac{\sum_{U \in S} AR(U) \cdot U}{\sum_{U \in S} U}. \quad (7.2)$$

The weighted acceptance ratio therefore provides a single value that summarizes the acceptance ratios over the utilization values in S , with each acceptance ratio weighted according to its corresponding utilization value. This allows the effect of changing a single parameter to be compared using the weighted acceptance ratio, rather than comparing acceptance ratios separately for every utilization value.

7.3 Simulation Setup

To evaluate the TaskShufflerGlobal algorithm, a simulator was developed to execute the generated task sets using three different scheduling algorithms. These consisted of a baseline global fixed-priority scheduler that makes scheduling decisions based solely on task priorities, a scheduler incorporating the TaskShufflerGlobal algorithm presented in Section 6, and a scheduler implementing the algorithm proposed in [8].

Since the work in [8] only provides pseudocode and employs a different schedulability analysis, the algorithm was implemented as faithfully as possible to the published description. For consistency, all generated task sets were analysed using the schedulability analysis proposed in Section 5.

Once jobs have been selected for execution, they are assigned to processors randomly, regardless of the scheduling algorithm used and irrespective of whether a job was previously executing on a particular processor. The resulting scheduling behaviour is analysed using entropy-based metrics to quantify the degree of variation introduced by the randomization algorithms.

As TaskShufflerGlobal is intended to operate proactively within the system model, the evaluation considers task sets without security-level transitions, effectively assuming that the system remains at the NS-level throughout the simulation.

The simulator was configured to test utilization levels from 10 to 80 percent on systems with 2, 4, and 8 processors. For utilization levels above 80 percent, generating valid task sets became increasingly difficult, especially for larger processor counts. The generated task sets followed the period constraints introduced in section 7.1, resulting in a maximum hyperperiod of 15360. The actual hyperperiod varied between runs depending on the specific task parameters produced during task generation.

During simulation, the executing task and its assigned processor were recorded at each time slot. After completing one hyperperiod, the entropy of the resulting schedule was computed. Each scheduler was run for up to 1000 hyperperiods or until the change in entropy between two consecutive runs was less than 0.001, ensuring that entropy had stabilised before recording the final value. For each utilization point and processor configuration, 100 independent task sets were generated and evaluated. To make the entropy values comparable across task sets, the final entropy was normalized by dividing it by the corresponding hyperperiod.

A second experiment investigated whether trying to hide idle processor slots could increase schedule entropy. In a real world system, an idle processor can reveal scheduling patterns to an external observer if they can, for example, measure the power consumption of a processor or observe memory accesses. To simulate a countermeasure, each idle slot was recorded as if one of ten lightweight filler tasks were executing, rather than leaving it idle. These filler tasks model benign background computation such as sorting routines, where different algorithms such as quicksort and mergesort could be cycled through to further obscure scheduling patterns. All other simulation parameters remained identical to the first experiment.

7.4 Runtime Metrics

Two metrics were considered when simulating the schedulers.

- Task-Level Entropy
- Processor-Level Entropy

The Task-Level Entropy metric corresponds to the uniprocessor schedule-entropy definition proposed in [5], shown in Equation 2.6. In the uniprocessor case, the slot entropy is computed from the distribution of tasks that appear in a given time slot across repeated executions. The total entropy is obtained by summing the slot entropies over the hyperperiod. This metric is well suited for quantifying variation in the time slots in which tasks execute across multiple schedule realizations. In the multiprocessor setting, however, each time slot may contain a set of tasks executing simultaneously on different processors, which this metric does not distinguish between.

The Processor-Level Entropy considers each time slot and processor individually. For a fixed processor p and slot t , the slot entropy is defined as:

$$H(t, p) = - \sum_{i=1}^N \Pr(t, p, \tau_i) \log_2 \Pr(t, p, \tau_i) \quad (7.3)$$

where $\Pr(t, p, \tau_i)$ is the probability that task τ_i executes on processor p at slot t . The processor-level scheduling entropy is obtained by summing the slot entropies over all slots and all processors:

$$H_{\text{proc}} = \sum_{p=1}^m \sum_{t=0}^{L-1} H(t, p). \quad (7.4)$$

Since the entropy values scale with both the number of processors and the length of the hyperperiod, all entropy results are normalized. Processor-Level Entropy is divided by mL , while Task-Level Entropy is divided by L , yielding average per-slot measures that allow fair comparison across different system sizes.

8

Results

This chapter presents the results of the schedulability analysis in Section 8.1, followed by the simulation results for the TaskShufflerGlobal algorithm in section 8.2.

8.1 Mixed-Trust Model Schedulability

Figure 8.1 shows the acceptance ratio of 1000 randomly generated task sets consisting of 15 application tasks, 3 monitoring tasks, and 3 security tasks. In this experiment, 70% of the application tasks are trusted, and their execution time increases by a factor of 1.25 at the HS-level. After a security level switch, three high-security tasks are introduced with a fixed combined utilization of 0.5, independent of the number of processors.

The results show that the acceptance ratio begins to decline at lower utilizations as the number of processors is reduced. With 8 processors, all task sets remain schedulable up to approximately 50% utilization. This threshold decreases to approximately 45% and below 40% on 4- and 2-processor platforms, respectively. This decline is more pronounced for systems with fewer processors, since the fixed HS utilization constitutes a larger proportion of the total available processing capacity. With 2 processors and HS utilization 0.5, the HS tasks use 25% of the total capacity while in the 8 processor case they account only for 6.25%.

Figure 8.2 compares the acceptance ratio for task sets with no trusted tasks, 70% trusted tasks, and fully trusted task sets on a 4-processor platform. Increasing the proportion of trusted tasks decreases the acceptance ratio, since more tasks keep executing after the security-level switch and incur the HS-level execution-time inflation. With no trusted tasks, all application tasks are suspended after the switch, and only the monitoring tasks and security tasks together with the injected high-security tasks execute at the HS-level. Using only trusted tasks, all application tasks continue executing with inflated HS-level execution times, resulting in the highest HS-level utilization. The 70% trust configuration represents a Mixed-Trust scenario in which most application tasks continue executing after the switch, while the remaining 30% are suspended.

These results illustrate the trade-off between security and schedulability in the Mixed-Trust model. Under the model assumptions, trusted tasks cannot be exploited by an attacker to gain access to the system. Increasing the trust ratio there-

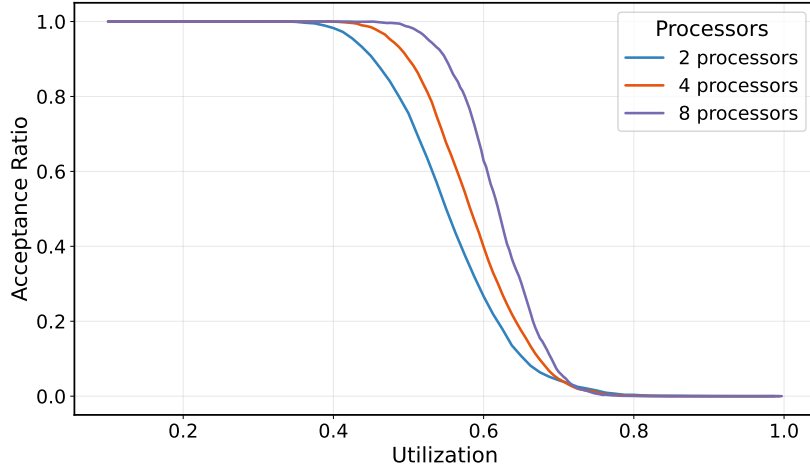


Figure 8.1: Acceptance ratio of randomly generated task sets composed of 15 tasks, 0.5 HS utilization, and a 1.25 scaling factor, using 2, 4, and 8 processors.

fore reduces the number of untrusted tasks that may serve as attack vectors and allows more application functionality to remain available following a security-level switch. However, because trusted tasks continue executing with inflated HS-level execution times, higher trust ratios also increase the post-switch workload and reduce schedulability. Conversely, lower trust ratios improve schedulability due to less work being performed at the HS-level, but at the cost of fewer trusted tasks, a less secure system, and less application functionality remaining active after the switch. The results demonstrate that improving the security posture of the system comes at the expense of reduced schedulability, while higher schedulability can only be achieved by accepting a lower level of security.

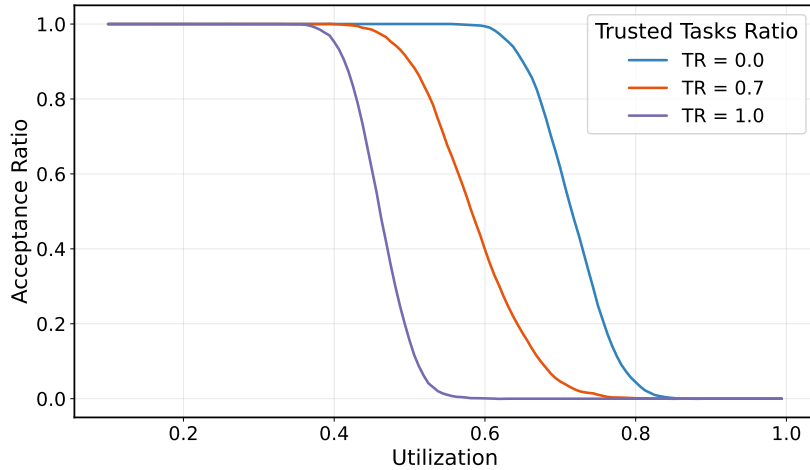


Figure 8.2: Acceptance ratio of randomly generated task sets with no trusted tasks, 70% trusted tasks, and a fully trusted task set on 4 processors.

The schedulability analysis results in Figures 8.1 and 8.2 align with prior work on mixed-criticality scheduling on multiprocessor platforms using OPA [10], but the

behaviour of the proposed Mixed-Trust model differs in several important ways. The MT model introduces additional high-security tasks and allows task periods to change after the level switch, which alters the post-switch workload. Consequently, the effect of the level switch on schedulability depends not only on the workload of the application tasks that remain active, but also on the additional workload introduced by the security mechanisms.

The model also assumes that trusted tasks cannot be exploited by an attacker. Nevertheless, a fully trusted configuration remains affected by false alarms. A false detection can trigger a transition to the HS-level even when no attack is present, causing the system to incur the additional HS-level workload. The accuracy of the monitoring mechanism is therefore important, since reducing false detections can avoid unnecessary transitions to the HS-level. If no false detections or actual attacks occur, the system remains at the NS-level and does not incur the additional HS-level workload during normal operation.

Figure 8.3 shows how the percentage of trusted tasks affects the acceptance ratio on each processor. The y-axis shows the weighted acceptance ratio.

The scaling factor for the execution time of trusted and security tasks is 1.25, while the period scaling factor for monitoring tasks is 0.75. The results show that increasing the trust ratio has a greater impact on systems with fewer processors.

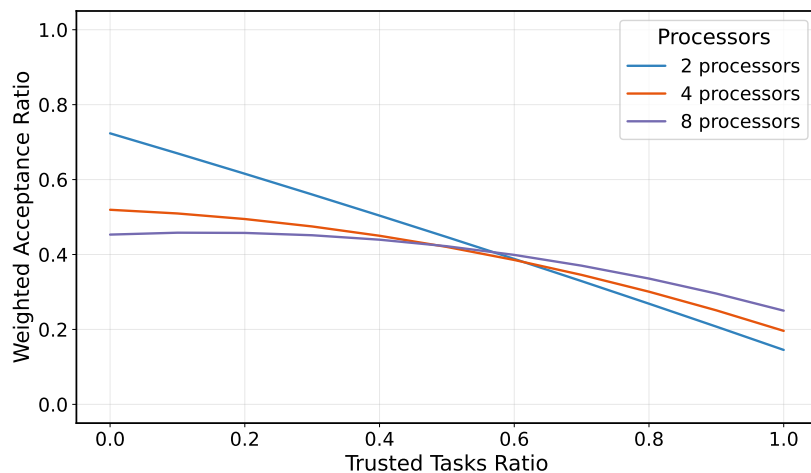


Figure 8.3: Weighted acceptance ratio comparing the effect of trust ratio on schedulability for 2, 4, and 8 processors.

The varying trust ratios reveal how processor count interacts with trust level. At low trust levels, systems with fewer processors initially accept more task sets, consistent with the reduced pessimism typically observed in multiprocessor mixed-criticality analysis. However, as the trust ratio increases, the fixed HS-level workload becomes a dominant factor. Consequently, the schedulability advantage of systems with fewer processors gradually disappears, and systems with higher processor counts eventually achieve higher acceptance ratios.

Figure 8.4 shows the effect on acceptance ratio when the execution time scale is

increased. All processor configurations appear to be affected similarly and approximately linearly by the increase in utilization due to the increased execution time at the HS-level.

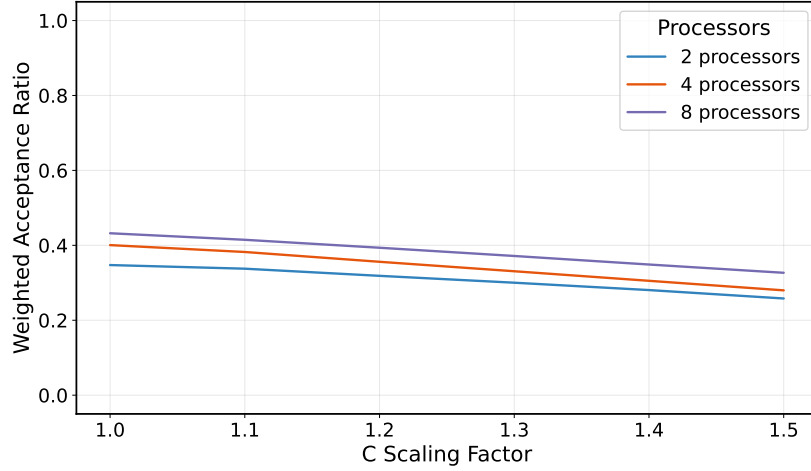


Figure 8.4: Weighted acceptance ratio comparing how the execution scaling factor of trusted tasks affects schedulability on 2, 4, and 8 processors.

The execution-time scaling factor directly increases the WCET of trusted tasks at the HS-level. Consequently, increasing the scaling factor increases the post-switch workload and reduces the acceptance ratio. The similar trend across processor counts indicates that the effect of the scaling factor is primarily determined by the additional execution demand introduced by the trusted tasks.

Figure 8.5 shows the effect of increasing the fixed amount of HS-level utilization introduced when the security level is increased. At low HS utilization, the 2-processor system has a higher acceptance ratio, consistent with the behaviour observed in the previous experiments. As the HS utilization increases, however, the fixed additional workload consumes an increasingly larger proportion of the available processing capacity on systems with fewer processors.

The HS-level utilization introduced by high-security tasks has a disproportionate effect on systems with fewer processors. Because the additional workload is fixed, its relative impact grows as the number of processors decreases. This explains the steeper decline in acceptance ratio for the 2-processor system as HS utilization increases.

The Mixed-Trust model allows tasks to execute at a higher frequency after a security-level switch. However, the randomly generated task sets used in this evaluation produced period-scaling ratios clustered around 0.75. As a result, the effect of period scaling was not isolated in the presented results. Therefore, while the analysis supports the use of different task periods across security levels, the specific contribution of period scaling to schedulability cannot be determined from these experiments alone.

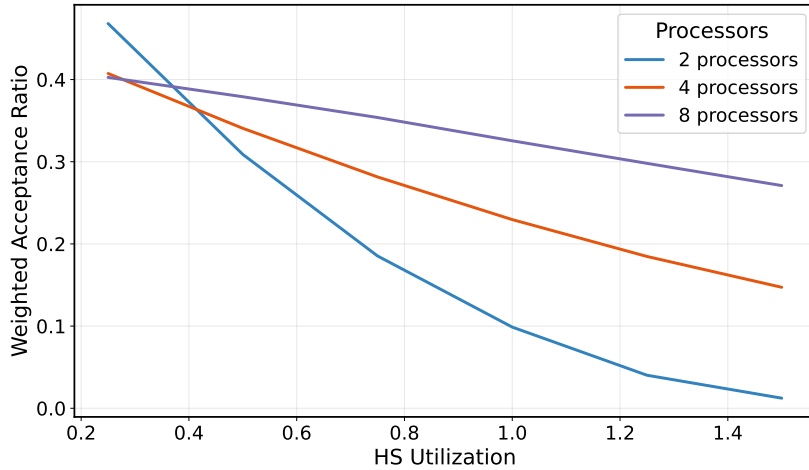


Figure 8.5: Weighted acceptance ratio comparing how HS utilization affects schedulability on 2, 4, and 8 processors.

Overall, the results show that processor count, trust ratio, execution-time scaling, and HS-level utilization all influence the schedulability of the Mixed-Trust system. The experiments demonstrate that the additional workload introduced at the HS-level has a particularly strong effect on systems with fewer processors, while increasing the amount of trusted functionality or its execution-time overhead further increases the post-switch workload.

8.2 TaskShufflerGlobal

Figure 8.6 shows the Task-Level Entropy results for TaskShufflerGlobal, denoted as *TSG*, and the TaskShuffler-based scheduler from [8], denoted as *TS-BK*.

The results show that *TSG* achieves higher Task-Level Entropy than *TS-BK* in all evaluated instances. The increase can be attributed to two changes in the scheduling algorithm: allowing a processor to remain idle rather than always executing a ready task, and allowing the candidate list to be extended beyond the first job with a non-positive budget. These changes provide additional opportunities for task execution to vary between scheduling decisions. An observation is that Task-Level Entropy is higher when fewer processors are used. Since the metric does not account for which processor a task executes on, the same time slot contributes less variation when multiple tasks can execute simultaneously. Task-Level Entropy increases when a task is observed in a larger number of different time slots. With more processors, the likelihood of a task executing when it is ready is greater, reducing the variation in the time slots in which the task executes.

The Task-Level Entropy exhibits a structural dependence on the number of processors. With fewer processors, the maximum number of tasks that can execute concurrently is limited, causing task execution to be distributed across more time slots. With 8 processors, up to 8 tasks can execute simultaneously, allowing exe-

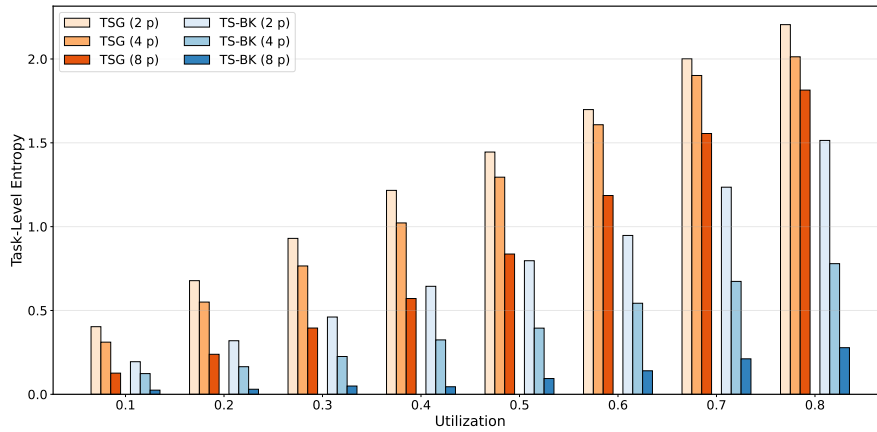


Figure 8.6: Average Task-Level Entropy

cution to be concentrated in fewer time slots, which reduces Task-Level Entropy. Consequently, higher Task-Level Entropy with 2 processors does not necessarily indicate greater overall scheduling randomness, but can partly result from the reduced parallelism available. The metric also does not distinguish between processor assignments. As a result, different schedules that execute the same tasks in the same time slots are considered equivalent, even when the tasks execute on different processors.

The Task-Level Entropy also increases with utilization in the multiprocessor experiments. In the uniprocessor case presented in [5], the metric exhibits a peak around 50% utilization, a behaviour that does not appear in the multiprocessor results. At low utilization, most time slots remain idle, limiting the number of distinct task sets that can appear and therefore keeping entropy low. This behaviour is consistent across both uniprocessor and multiprocessor systems. At high utilization, the uniprocessor case shows reduced entropy because the number of jobs with a positive inversion budget decreases, and the remaining budgets become small, limiting the scheduler’s ability to introduce variation. In the multiprocessor case, however, at least the m highest-priority tasks have a positive inversion budget when it is calculated offline. If their individual utilizations are modest, they retain substantial inversion budgets even at high overall utilization. At each scheduling decision, up to m jobs can therefore have a positive budget, which appears to allow *TSG* to continue introducing variation in task-execution patterns and prevent the decline in entropy observed in the uniprocessor case.

Figure 8.7 shows the comparison of *TSG* and *TS-BK* using Processor-Level Entropy. Again, *TSG* achieves higher entropy than *TS-BK*, although the difference is smaller than for Task-Level Entropy. In contrast to Task-Level Entropy, Processor-Level Entropy increases with the number of processors.

Figure 8.8 shows the comparison of *TSG* and the non-randomized global scheduler using Processor-Level Entropy, this is denoted as *Global* in the figure. For 2 and 4 processors, there is a large increase in entropy compared to the baseline case, while with more processors the difference becomes smaller, particularly at low utilization. Processor-Level Entropy captures both when a task executes and on which processor

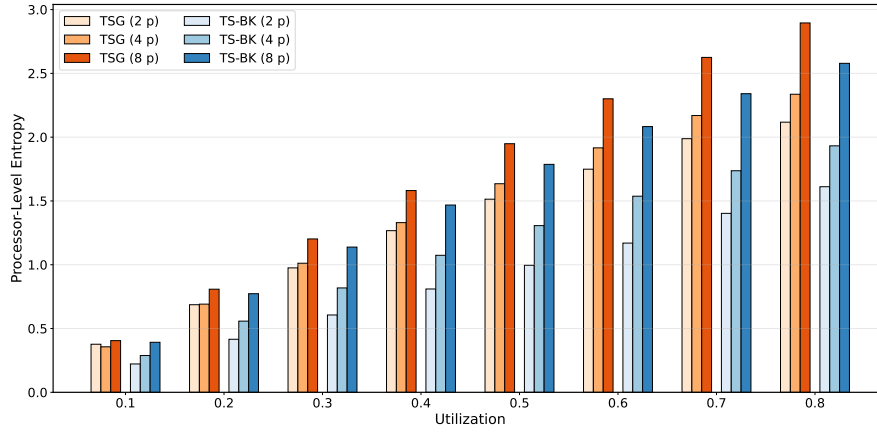


Figure 8.7: Average Processor-Level Entropy of *TSG* and the *TS-BK* algorithms on 2, 4, and 8 processors

it executes. With more processors, a larger proportion of the entropy can therefore arise from the processor assignment. An important distinction is that Task-Level Entropy for the deterministic global scheduler without randomization is 0, since there is no variation in the time slot in which a task executes. However, Processor-Level Entropy is still introduced because the processor executing a job is selected randomly. The results show that, when processor assignments are considered separately, systems with more processors consistently exhibit higher entropy, even after normalization by the number of processors.

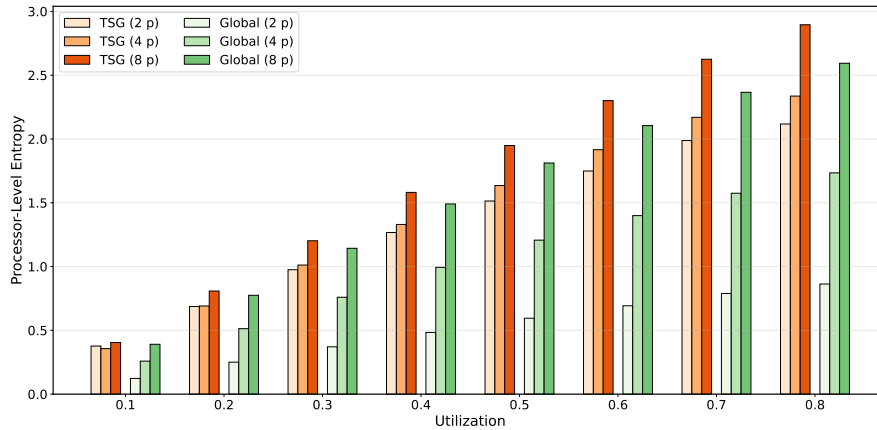


Figure 8.8: Average Processor-Level Entropy

Task-Level Entropy and Processor-Level Entropy must be interpreted together, as neither metric alone captures the full picture. The Task-Level Entropy reflects randomness in when a task executes. For the deterministic scheduler this is zero, as task timing is fixed. The Processor-Level Entropy, on the other hand, captures the average entropy per slot per processor, reflecting randomness in which processor executes a given task. This distinction becomes important when comparing the 8-processor case: although the Processor-Level Entropy of the deterministic scheduler

approaches that of GlobalTaskShuffler, the sources of entropy differ entirely. In GlobalTaskShuffler entropy arises from both task timing and processor assignment, whereas in the deterministic scheduler all entropy stems exclusively from processor assignment.

The Processor-Level Entropy observed even in the deterministic global scheduler suggests that global scheduling provides a limited form of inherent protection against attacks that rely on predicting the exact processor on which a task will execute. Because the simulator resolves ties by randomly selecting among the available processors, the mapping from tasks to processors becomes unpredictable across executions. This spatial unpredictability may disrupt both anterior and posterior attacks that depend on correlating cache state, memory access patterns, or execution timing with a specific processor. It indicates that global scheduling naturally introduces a degree of uncertainty that may complicate processor-targeted side-channel attacks. While prior work has shown that schedule randomization alone cannot prevent all schedule-based attacks, the spatial unpredictability introduced by migration in multiprocessor systems provides an additional dimension of uncertainty that may further hinder such attacks.

Figure 8.9 shows a comparison between all the three different schedulers used with 4 processors. An interesting observation is that increasing the number of processors appears to have a larger effect on the measured Processor-Level Entropy than the introduction of a randomization algorithm. This suggests that processor assignment itself contributes substantially to entropy in global scheduling systems.

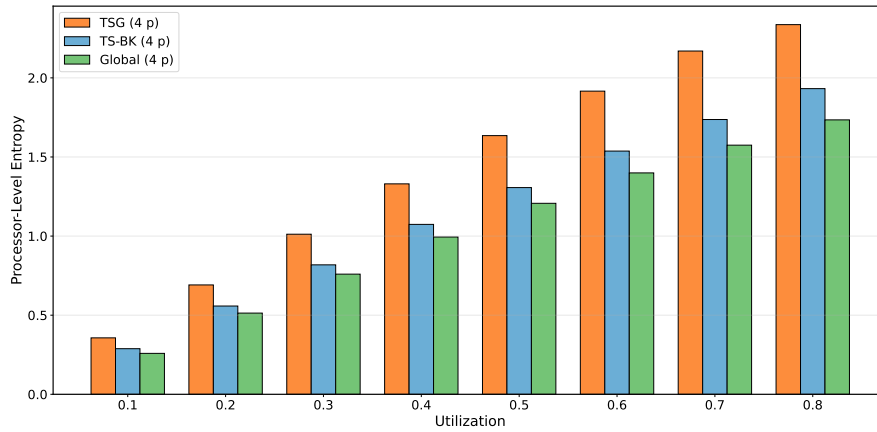


Figure 8.9: Average Processor-Level Entropy comparison of *TSG*, *TS-BK* and the non randomized global scheduler using 4 processors

Figure 8.10 shows the average entropy of *TSG* with and without the addition of tasks executing in what would otherwise be an idle time slot. Denoting the scheduler with this addition as *TSG+*, it is clear that this modification significantly increases the Processor-Level Entropy.

It is important to note that the increase in entropy arises solely from observing processor activity within individual time slots. The execution pattern of the actual system tasks remains unchanged. The effect is most pronounced at low utilization

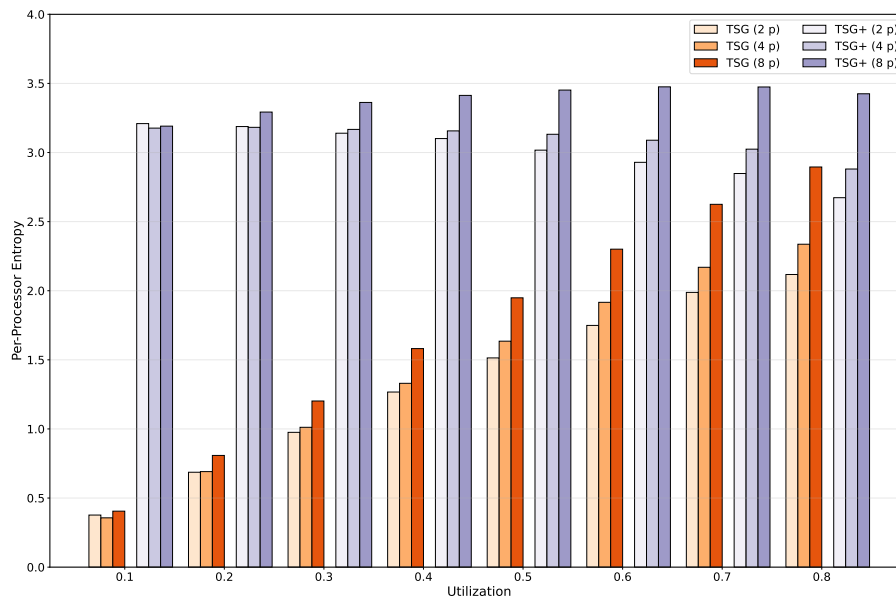


Figure 8.10: Average entropy of *TSG* and *TSG+* using 10 additional tasks executing when normally it should be idle.

levels, where processors would otherwise spend a large proportion of time idle. Introducing additional activity into these idle slots obscures the idle-time pattern, making it more difficult for an attacker to predict when an attack may be most effective.

Overall, the results demonstrate that the *TSG* algorithm introduces additional scheduling randomness compared with the non-randomized global scheduler and achieves higher measured entropy than the previous multiprocessor implementation presented in [8]. The results also show that the measured entropy depends strongly on the choice of metric and the number of processors. Task-Level Entropy primarily captures variation in when tasks execute, while Processor-Level Entropy additionally captures variation in processor assignment. Therefore, both metrics are useful for evaluating different aspects of scheduling randomness in global multiprocessor systems.

9

Conclusion

The aim of this thesis was to design a security-aware task model for globally scheduled multiprocessor real-time systems. Building on the mixed-criticality paradigm, the proposed Mixed-Trust model assigns tasks trust levels rather than traditional criticality levels, with untrusted tasks suspended upon detection of a potential attack. The model defines a reactive security mechanism in which monitoring tasks trigger a transition to a higher security level, freeing computational resources for trusted tasks and enhanced security measures. In addition, a proactive defense against schedule-based attacks was developed through TaskShufflerGlobal, a global multiprocessor implementation of the TaskShuffler algorithm that introduces scheduling randomness to improve resilience against side-channel attacks. The evaluation focuses on the schedulability properties of the mixed-trust model and the entropy characteristics of TaskShufflerGlobal and its extensions.

The evaluation of the mixed-trust model shows that it preserves the core behaviour of mixed-criticality scheduling while introducing new dynamics driven by the addition of high-security tasks and varying periodicityan aspect previously unexplored in fixed- priority global scheduling. In particular, the fixed HS-mode workload and the continued execution of trusted tasks after a security-level switch may make systems with fewer processors more difficult to schedule than systems with many processors, highlighting an important design consideration for mixed-trust deployments. These results indicate that trust assignment and processor count must be carefully balanced to ensure that the system can sustain a level switch without violating timing guarantees.

The simulation results demonstrate that TaskShufflerGlobal successfully introduces both temporal and spatial unpredictability into globally scheduled multiprocessor systems. While the Task-Level and Processor-Level Entropy metrics have limited resolution for distinguishing fine-grained differences between schedules, they consistently show that TaskShufflerGlobal increases entropy compared to the previous work of [8]. The incidental spatial unpredictability introduced by global schedulingstemming from nondeterministic processor assignment further suggests that migration may complicate processor-targeted attacks, complementing the temporal variation introduced by the shuffling mechanism.

The execution of additional tasks during otherwise idle time slots was shown to significantly increase entropy, making it a worthwhile enhancement provided that it can be implemented without compromising real-time guarantees. Although these tasks

do not alter the execution patterns of the application tasks themselves, they reduce the predictability of processor activity by obscuring idle intervals. An additional potential benefit is that the execution of these tasks may modify cache contents, thereby reducing the amount of information that could be inferred through cache-based observations and potentially limiting information leakage between tasks.

Taken together, these findings show that the mixed-trust model and TaskShufflerGlobal address two complementary aspects of security: proactive disruption of schedule-based attacks and reactive containment of untrusted tasks when an attack is detected. Although the proposed mechanisms introduce additional overhead and rely on assumptions about monitoring accuracy and system behaviour, the results indicate that meaningful security benefits can be achieved without compromising schedulability under realistic conditions. These results illustrate the potential of the proposed model as a foundation for further exploration of security-aware scheduling in multiprocessor real-time systems.

9.1 Future Work

Future work should explore more expressive entropy metrics capable of capturing fine-grained differences between schedules, as the current metrics have limited resolution for distinguishing subtle variations in task-execution patterns. Such metrics could also incorporate an attacker-aware perspective, enabling a more direct assessment of the effectiveness of scheduling randomization against specific classes of adversaries.

Evaluating the runtime overhead of TaskShufflerGlobal on real hardware is also an important next step, as the present evaluation focuses exclusively on simulated behaviour. Experimental validation on representative multiprocessor platforms would provide insight into the practical cost of the proposed randomization mechanisms and their suitability for deployment in real-world systems.

Since WCET is a pessimistic parameter, the actual execution time of a job is often shorter than its WCET. Future work could therefore investigate whether the slack resulting from early job completion can be reclaimed by the randomization algorithm. Reclaimed slack could be used to introduce additional scheduling choices or preemptions, potentially increasing variation in task-execution patterns and, consequently, the measured scheduling entropy.

Bibliography

- [1] R. I. Davis and B. A., “Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems,” *Real-Time Systems*, vol. 47, pp. 1–40, 2011. DOI: 10.1007/s11241-010-9106-5.
- [2] C. Chen, S. Mohan, and R. Bobba, “Schedule-based side-channel attack in fixed-priority real-time systems,” University of Illinois at Urbana-Champaign, Tech. Rep., 2015. [Online]. Available: <http://hdl.handle.net/2142/88344>.
- [3] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash, “A novel side-channel in real-time schedulers,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 90–102. DOI: 10.1109/RTAS.2019.00016.
- [4] J. Son and Alves-Foss, “Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems,” in *2006 IEEE Information Assurance Workshop*, 2006, pp. 361–368. DOI: 10.1109/IAW.2006.1652117.
- [5] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, “Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems,” in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–12. DOI: 10.1109/RTAS.2016.7461362.
- [6] S. Mohan, M. K. Yoon, R. Pellizzoni, and R. Bobba, “Real-time systems security through scheduler constraints,” in *2014 26th Euromicro Conference on Real-Time Systems*, 2014, pp. 129–140. DOI: 10.1109/ECRTS.2014.28.
- [7] M. Bertogna and M. Cirinei, “Response-time analysis for globally scheduled symmetric multiprocessor platforms,” in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007, pp. 149–160. DOI: 10.1109/RTSS.2007.31.
- [8] H. Baek and C. M. Kang, “Scheduling randomization protocol to improve schedule entropy for multiprocessor real-time systems,” *Symmetry*, vol. 12, no. 5, 2020, ISSN: 2073-8994. DOI: 10.3390/sym12050753. [Online]. Available: <https://www.mdpi.com/2073-8994/12/5/753>.
- [9] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, “Securecore: A multicore-based intrusion detection architecture for real-time embedded systems,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 21–32. DOI: 10.1109/RTAS.2013.6531076.
- [10] R. M. Pathan, “Schedulability analysis of mixed-criticality systems on multiprocessors,” in *2012 24th Euromicro Conference on Real-Time Systems*, 2012, pp. 309–320. DOI: 10.1109/ECRTS.2012.29.

- [11] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007, pp. 239–243. DOI: 10.1109/RTSS.2007.47.
- [12] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 34–43. DOI: 10.1109/RTSS.2011.12.
- [13] J. Wynn et al., *Threat assessment & remediation analysis (tara)*, Accessed: 2026-05-02, 2011. [Online]. Available: <https://www.mitre.org/sites/default/files/2021-10/pr-11-4982-tara-methodology-and-description.pdf>.
- [14] M. Benyahya, T. Lenard, A. Collen, and N. A. Nijdam, "A systematic review of threat analysis and risk assessment methodologies for connected and automated vehicles," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23, Benevento, Italy: Association for Computing Machinery, 2023, ISBN: 9798400707728. DOI: 10.1145/3600160.3605084. [Online]. Available: <https://doi.org/10.1145/3600160.3605084>.
- [15] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 5 1993. DOI: 10.1049/sej.1993.0034. eprint: <https://digital-library.theiet.org/doi/pdf/10.1049/sej.1993.0034>. [Online]. Available: <https://digital-library.theiet.org/doi/abs/10.1049/sej.1993.0034>.
- [16] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973, ISSN: 0004-5411. DOI: 10.1145/321738.321743. [Online]. Available: <https://doi.org/10.1145/321738.321743>.
- [17] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, Jan. 1986, ISSN: 0010-4620. DOI: 10.1093/comjnl/29.5.390. eprint: <https://academic.oup.com/comjnl/article-pdf/29/5/390/1314410/290390.pdf>. [Online]. Available: <https://doi.org/10.1093/comjnl/29.5.390>.
- [18] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, Oct. 2011, ISSN: 0360-0300. DOI: 10.1145/1978802.1978814. [Online]. Available: <https://doi.org/10.1145/1978802.1978814>.
- [19] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Oper. Res.*, vol. 26, no. 1, pp. 127–140, Feb. 1978, ISSN: 0030-364X. DOI: 10.1287/opre.26.1.127. [Online]. Available: <https://doi.org/10.1287/opre.26.1.127>.
- [20] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, 2001, pp. 193–202. DOI: 10.1109/REAL.2001.990610.
- [21] N. Audsley, "On priority assignment in fixed priority scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001, ISSN: 0020-0190. DOI: [https://doi.org/10.1016/S0020-0190\(01\)00039-9](https://doi.org/10.1016/S0020-0190(01)00039-9).

- [//doi.org/10.1016/S0020-0190\(00\)00165-4](https://doi.org/10.1016/S0020-0190(00)00165-4). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0020019000001654>.
- [22] B. Andersson and J. Å. Jönsson, “Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling,” in *IEEE Real-Time Systems Symposium*, 2000. [Online]. Available: <https://api.semanticscholar.org/CorpusID:11644213>.
 - [23] *ISO 26262-1:2018 road vehicles – functional safety*, International Organization for Standardization, 2018.
 - [24] *DO-178C: Software considerations in airborne systems and equipment certification*, RTCA, 2011.
 - [25] M. Nasri, T. Chantem, G. Bloom, and R. M. Gerdes, “On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 103–116. DOI: 10.1109/RTAS.2019.00017.
 - [26] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.
 - [27] N. Guan, M. Stigge, W. Yi, and G. Yu, “New response time bounds for fixed priority multiprocessor scheduling,” in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, ser. RTSS ’09, USA: IEEE Computer Society, 2009, pp. 387–397, ISBN: 9780769538754. DOI: 10.1109/RTSS.2009.11. [Online]. Available: <https://doi.org/10.1109/RTSS.2009.11>.
 - [28] S. Baruah, “Schedulability analysis of mixed-criticality systems with multiple frequency specifications,” in *Proceedings of the 13th International Conference on Embedded Software*, ser. EMSOFT ’16, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2016, ISBN: 9781450344852. DOI: 10.1145/2968478.2968488. [Online]. Available: <https://doi.org/10.1145/2968478.2968488>.
 - [29] S. Baruah, T. Chantem, N. Fisher, and F. Raadia, “A scheduling model inspired by security considerations,” in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, 2023, pp. 32–41. DOI: 10.1109/ISORC58943.2023.00016.
 - [30] S. Baruah, “The limited-preemption uniprocessor scheduling of sporadic task systems,” in *17th Euromicro Conference on Real-Time Systems (ECRTS’05)*, 2005, pp. 137–144. DOI: 10.1109/ECRTS.2005.32.
 - [31] D. de Niz et al., “Mixed-trust computing for real-time systems,” in *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2019, pp. 1–11. DOI: 10.1109/RTCSA.2019.8864566.
 - [32] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, “A design-space exploration for allocating security tasks in multicore real-time systems,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 225–230. DOI: 10.23919/DATE.2018.8342007.
 - [33] K. Jiang, L. Batina, P. Eles, and Z. Peng, “Robustness analysis of real-time scheduling against differential power analysis attacks,” in *2014 IEEE Com-*

- puter Society Annual Symposium on VLSI*, 2014, pp. 450–455. DOI: 10.1109/ISVLSI.2014.11.
- [34] G. H. Kim and E. H. Spafford, “The design and implementation of tripwire: A file system integrity checker,” in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, ser. CCS ’94, Fairfax, Virginia, USA: Association for Computing Machinery, 1994, pp. 18–29, ISBN: 0897917324. DOI: 10.1145/191177.191183. [Online]. Available: <https://doi.org/10.1145/191177.191183>.
 - [35] L. Zhang, P. Lu, F. Kong, X. Chen, O. Sokolsky, and I. Lee, “Real-time attack-recovery for cyber-physical systems using linear-quadratic regulator,” *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, Sep. 2021, ISSN: 1539-9087. DOI: 10.1145/3477010. [Online]. Available: <https://doi.org/10.1145/3477010>.
 - [36] P. Emberson, R. Stafford, and R. Davis, “Techniques for the synthesis of multiprocessor tasksets,” *WATERS’10*, Jan. 2010.
 - [37] R. M. Pathan, “Improving the Schedulability and Quality of Service for Federated Scheduling of Parallel Mixed-Criticality Tasks on Multiprocessors,” in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, S. Altmeyer, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 106, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 12:1–12:22, ISBN: 978-3-95977-075-0. DOI: 10.4230/LIPIcs.ECRTS.2018.12. [Online]. Available: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2018.12>.