Security-Aware Scheduling of Real-Time Tasks on Multi-core Processors

Protecting cyber-physical systems with randomized execution schedules

Master's thesis in Computer science and engineering

Liam Cotton

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Security-Aware Scheduling of Real-Time Tasks on Multi-core Processors

Protecting cyber-physical systems with randomized execution schedules

Liam Cotton

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Security-Aware Scheduling of Real-Time Tasks on Multi-core Processors
Protecting cyber-physical systems with randomized execution schedules
Liam Cotton

© Liam Cotton, 2025.

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

Master's Thesis 2025
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 LaTeX
Gothenburg, Sweden 2025

Security-Aware Scheduling of Real-Time Tasks on Multi-core Processors
Protecting cyber-physical systems with randomized execution schedules
Liam Cotton
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern real-time systems are increasingly exposed to timing-based security threats due to their predictable task scheduling. When scheduling tasks for real-time execution, a predictable execution pattern is needed to ensure all tasks will meet their deadlines. A common practice is to employ a fixed-priority scheduler, a deterministic scheduling algorithm always choosing the same task to execute every time it's given the same conditions. Schedule-based attacks exploit this determinism, enabling adversaries to manipulate or extract sensitive information by aligning their execution with critical tasks. To counter this, schedule randomization has emerged as a potential solution, introducing controlled unpredictability into task execution.

This thesis investigates the application of schedule randomization in multi-core real-time systems, particularly when tasks are pre-allocated to specific cores. The study builds upon TaskShuffler, an already existing algorithm that introduces randomness into the previously deterministic fixed priority scheduler. This algorithm, designed for single-core systems, is now extended for multi-core use. Further, we examine techniques to mitigate or circumvent schedule-based attacks targeting multi-core systems. We also extend the concept of *schedule entropy*, a "randomness" metric, to better suit multi-core systems, as well as introduce new security-aware metric to capture the risk of common types of targeted attacks. We evaluate the security and performance impact of our methods by by simulating tasks execution on multi-core processors under different task sets and configurations. This provides insights into how core assignment and priority relations affect the system's exposure to schedule-based attacks. Such insights may help the system designer to strengthen the security of the systems by allocating or not allocating certain tasks to certain cores at design time.

Keywords: Real time systems, security, schedule based attacks, cyber physical systems, Computer, science, computer science, engineering, project, thesis

Acknowledgements

I would like to express my gratitude to these individuals for mentoring and supporting me in completing this project.

To my supervisor, Risat, for all our intense weekly meeting, providing both inspiration and direction. You are the most thorough proofreader I've ever met.

To my co-supervisor, Xiuqi, for helping me run my simulations on the cluster.

To my examiner, Jan, retiring shortly, for joining my final presentation on his day off.

To my fellow engineering students in the Chalmers choir, for helping me push through tough times.

Liam Cotton, Gothenburg, 2025-06-18

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Previous work 2
1.2 Project aim 4
1.2.1 Restrictions 4
1.2.2 Challenges 4
1.3 Contributions 5
1.4 Thesis outlook 6

2 Theory **7**
2.1 Background 7
2.1.1 The Fixed-Priority Rate-Monotonic scheduler 7
2.1.2 Feasibility Tests 9
2.1.2.1 Response-Time Analysis (Exact Test) 9
2.1.2.2 Response-Time Analysis with Jitter 10
2.1.2.3 Response-Time Analysis during execution 11
2.1.3 Optimal Priority Assignment (OPA) 12
2.1.4 Partitioned Scheduling in Multi-core Systems 14
2.2 TaskShuffler 15
2.2.1 Measuring Randomness with Schedule Entropy 17
2.2.2 Improving taskshuffler 19
2.3 Metrics 20
2.3.1 Multi-core Schedule Entropy 20
2.3.2 Security-Aware Metrics 23
2.3.3 Terminology 25
2.4 Task Set Generation 25

3 Methods **27**
3.1 System and Threat Model 27
3.2 System Model 27
3.2.1 Threat Model 28
3.3 Protection strategies 29
3.3.1 TaskShuffler 29

3.3.2 Migration 29
3.3.3 Reprioritizing 30
3.3.4 Pushback 30
3.4 Simulations 31
3.4.1 Task set generation 31
3.4.2 Exhaustive test 31
3.5 Metrics 32

4 Results 33
4.1 Partition Algorithms and Priority Orderings 33
4.1.1 Exhaustive test 33
4.1.2 Custom Partitioning Algorithms 37
4.1.3 Large scale test: Comparative Analysis of Allocation Algorithms 39
4.2 Mitigation strategies 43
4.2.1 RePri - Reprioritizing tasks 43
4.2.2 Migration 45
4.2.3 Uni-core mitigation 47
4.2.3.1 Budget 47
4.2.3.2 Pushback 48
4.2.3.3 Mitigation without using pushback 49
4.2.3.4 Combining all mitigation strategies 50
4.2.4 Multi-core mitigation 52

5 Conclusion 57
5.1 Discussion 57
5.2 Conclusion 58
5.3 Future Work 58

Bibliography 61

List of Figures

2.1 Example of an execution schedule generated by a fixed-priority scheduler. 8
2.2 Response time R'_i of task τ_i after a potential migration or priority change, showing remaining execution of a high-priority task τ_k and its subsequent arrivals. 11
2.3 Execution schedule across several hyperperiods without TaskShuffler. 16
2.4 Shuffled execution of tasks across three hyperperiods using TaskShuffler. 16
2.5 Illustration of high-priority task interference within $D_i + V_k + J_k$. . . 20

4.1 Average horizontal schedule entropy across different priority orderings compared to the optimal ordering. 34
4.2 Horizontal schedule entropy for First-Fit partitioning under different task picking orders. 34
4.3 Horizontal schedule entropy for Worst-Fit partitioning under different task picking orders. 35
4.4 Vertical schedule entropy for First-Fit partitioning under different task picking orders. 36
4.5 Vertical schedule entropy for Worst-Fit partitioning under different task picking orders. 36
4.6 Horizontal schedule entropy for custom partitioning algorithms, compared to the optimal partition. 38
4.7 Vertical schedule entropy for custom partitioning algorithms, compared to the optimal partition. 39
4.8 Schedule entropy for BF and FF algorithms under different task picking orders. 40
4.9 BF and FF show similar anterior attack resiliency regardless of task picking order. 40
4.10 Schedule entropy of WF under different task picking orders. 41
4.11 Task picking order has low influence on anterior attack security in WF. 41
4.12 Comparison of schedule entropy across traditional and custom allocation strategies. 42
4.13 Anterior attack security is highest for WF, lowest for BF. Custom strategies fall in between, balancing compactness and spread. 42
4.14 Posterior attack risk shows minimal dependence on allocation algorithm. 43
4.15 Deviating from RM priority ordering reduces schedule entropy, with more frequent reprioritization yielding slightly greater losses. 44

4.16 Reprioritization has only a minor effect on attack risk, with noticeable differences limited to high-utilization task sets. 44
4.17 Regular task migration significantly reduces the risk of horizontal attacks. 45
4.18 Migration has limited impact on system-wide attacks that span across cores. 46
4.19 Effect of task migration on schedule entropy. Tasks allocated using WF-DU. 47
4.20 Enhanced budget calculations have negligible impact on attack resiliency in single-core systems. 48
4.21 Enhanced budget calculations lead to increased schedule entropy for task sets with high utilization. 48
4.22 Pushback has limited effect on posterior attack security and can, in some cases, reduce it. 49
4.23 Pushback significantly reduces schedule entropy by promoting predictable execution patterns. 49
4.24 Impact of mitigation strategies on attack resiliency. 50
4.25 Most mitigation strategies reduce schedule entropy compared to budget-optimized TaskShuffler. 50
4.26 Combination of pushback and regular reprioritization significantly improves posterior attack security. 51
4.27 Overall improvement in attack resiliency when combining mitigation strategies. 51
4.28 Improved budget calculations do not fully offset the entropy loss introduced by other mitigation strategies. 52
4.29 Reprioritization improves horizontal anterior security only when used alongside migration. 52
4.30 Pushback significantly improves horizontal posterior security only when combined with reprioritization. 53
4.31 Slight improvement in vertical posterior security. Migration contributes minimally. 53
4.32 Mitigation strategies show little to no improvement in vertical anterior security. 54
4.33 Without mitigation, security rapidly degrades as the number of untrusted tasks increases. 54
4.34 Multi-core mitigation strategies mitigate the drop in schedule entropy. 55

List of Tables

2.1 Task assignments across four hyperperiods at time t_0 for a 4-core system. Empty slots are represented using the $\tau_i^{\emptyset}$ notation. 22
2.2 Empirical frequencies ($\#_i$), estimated probabilities (P_i), and contribution to time slot entropy ($-H_\Gamma(S_t)_i$) for each task at $t = t_0$. 22

1
Introduction

In an era of rapid technological evolution, cyber-physical systems (CPS), spanning autonomous vehicles, smart grids, and industrial automation, have become critical to modern infrastructure. These systems rely on real-time computing to meet stringent timing constraints, where even microsecond delays can jeopardize safety and functionality. As the complexity of workloads continues to rise, CPS increasingly leverage multi-core processors to meet performance demands, emphasizing the need to maintain both timeliness and security to ensure reliability and resilience[1][2].

This increased computational sophistication also brings greater exposure to threats. The connection of real-time systems to external devices and the physical environment opens pathways for cyber-attacks that exploit predictable timing behavior. Attackers can not only infer sensitive system information through timing inference techniques but also manipulate it to disrupt operations or compromise integrity. Such attacks can have severe consequences, including safety hazards, environmental harm, and substantial financial loss. Real-world incidents such as the 2015 cyberattack on Ukraine's power grid and the 2010 Stuxnet worm, which targeted programmable logic controllers (PLCs) in Iran's nuclear facilities, highlight the devastating potential of these vulnerabilities[3][4].

When building dependable real-time systems, one must be certain that the system will be able to meet the deadlines of all incoming tasks. To ensure this, time-predictability is required for the system to be analyzed and proven to be dependable. Scheduling a task set where the tasks are scheduled based on a priority-based scheduler makes the systems thus makes the system deterministic, leaving it vulnerable to attacks exploiting the fact that it becomes easy to predict what task is to be run when. For instance, this might help an attacker compromise a low-security task. With access to the system, the attacker can now compromise the high-security tasks.

Many cyber-physical systems (CPS) operate under real-time constraints, where correct system behavior depends not only on logical results but also on the timing of those results. In such systems, control and monitoring operations are typically modeled using *periodic tasks*: tasks that are released at regular intervals and must complete within a defined deadline. Whether or not all deadlines are met depends heavily on the scheduling strategy employed by the system. A widely adopted class of schedulers, particularly in industry, is *fixed-priority scheduling*, favored for its simplicity, analyzability, and implementation efficiency.

Fixed-priority schedulers assign a static priority to each task prior to execution. Under this model, higher-priority tasks are allowed to preempt lower-priority tasks if the processor is busy. For example, in a rate-monotonic (RM) scheduler, tasks with shorter periods are given higher priority. Tasks are released periodically, and the period of each task may differ. If all tasks are initially released during the same time, there will be a point in the future, equal to the least common multiple of the task periods, where they are released at the same time again. At this point, referred to as the hyperperiod of the task, the scheduler will make the exact same scheduling decision as it did one hyperperiod ago. This creates a *deterministic* schedule. Such determinism enables formal analysis and schedulability tests to guarantee that all tasks meet their deadlines, that is, the system is *predictable*.

While determinism facilitates verification, it also introduces a security risk: an attacker who observes the system over time can infer future execution patterns and exploit them. Once the execution pattern is learned, an attacker may align their malicious task to execute alongside or just before a target task, potentially compromising sensitive operations.

To counter this, one can introduce *randomization* into the execution order of tasks. This means that, under certain conditions, the scheduler may choose to execute a lower-priority task like τ_2 before the higher-priority task τ_1, or even interleave parts of τ_2 before and after τ_1. Such behavior breaks the strict deterministic ordering, making the overall task execution sequence less predictable and harder for an attacker to exploit.

However, this randomization must be applied carefully. High-priority tasks are typically time-critical and must not miss their deadlines. Thus, before allowing a lower-priority task to delay a higher-priority one, the scheduler must verify that the higher-priority task has enough time left until its deadline to allow for other low-priority tasks to execute before it. This ensures that the system remains predictable in terms of deadline adherence, even as it becomes less deterministic in task ordering.

By strategically randomizing task execution order while preserving deadline guarantees, real-time systems can increase their resilience to timing-based attacks without sacrificing safety or correctness. This approach has been applied to uni-core systems, but its limitation to single-core architectures leaves an important gap in protecting multi-core systems. As multi-core systems become increasingly common in modern cyber-physical systems (CPSs) [2], this thesis aims to address this gap by developing a security-aware scheduling algorithm designed for multi-core environments, as well as proposing new metric to compute the effectiveness of randomization considering different models of the attacker.

1.1 Previous work

The TaskShuffler algorithm, introduced in prior research [5], represents an advancement in addressing security challenges in real-time systems. By incorporating randomization into the rate monotonic scheduler, TaskShuffler reduces the determinism of task schedules. This helps counter adversaries exploiting such determinism,

while still maintaining real-time guarantees.

In TaskShuffler[5], *schedule entropy* (presented in more detail in 2.2.1) is used to quantify how deterministic an execution schedule is. The entropy is computed based on the probabilities of different tasks being executed at any particular time, yielding a high score if tasks are equally likely to execute, signifing an undeterministic schedule.

Multi-core systems are more complex, and transferring the concept of schedule entropy to a multi-core setting would require special care accurately quantify the security of the entire system. Depending on the type of attacker considered, the system can be compromised by a single low performing core serving as an entry point for an attacker. In other cases, where an attacker uses a shared system resource and is unable to distinguish which core a task runs on, a different metric would be more appropriate.

TaskShuffler was developed for uni-core systems, and no research of schedule randomization has been done in the multi-core domain. The most basic way to apply TaskShuffler on a multi-core system would be to statically assign tasks to different cores and let each core run TaskShuffler independently for the runtime of the system. This would randomize the execution of each core individually as TaskShuffler has previously done on uni-core processors. Continuing to develop on this idea, this thesis will further randomize the schedules by allowing tasks to be reassigned to other cores during execution. When tasks move between cores, also known as *migration*, ensuring that all tasks will meet their deadline in the new configuration is not arbitrary, why such reassignments must be performed only when the resulting task allocation is proven to continue to meet all deadlines.

As shown in the TaskShuffler paper [5], the schedule entropy after randomization was improved the most for task sets with a medium load (utilization closer to 50% rather than extremely high or low). This suggests that if tasks are assigned to the cores in such a way that it evens out the load it might result in an over-all high entropy schedule. When extending TaskShuffler to multi-processor systems, there is an opportunity to take advantage of the task assignment process to maximize the schedulers' potential to randomize task executions. Task assignment is not a trivial problem, it is in fact an NP hard problem [6], and understanding how it affects the randomness of the generated schedule is an important challenge which is researched in this thesis.

Nasri et al. [7] critically evaluated schedule randomization (with focus on TaskShuffler) as a defense mechanism against different schedule-based attacks in real-time systems. These attacks exploit the deterministic nature of real-time scheduling to infer or interfere with the execution of victim tasks, often relying on precise timing relationships. The work in [7] formally defined different attack types: Anterior, Posterior, Pincer, and Concurrent, each depending on when the attacker executes relative to the victim. It was shown that randomizing schedules can, in some cases, increase the attack success rate rather than mitigating it. The authors of [7] advocate for attack-aware security metrics and caution that randomization alone is not sufficient without considering system-level constraints and attacker capabilities. In response, this thesis introduces an attack-aware metric, measuring how resilient the system is

against the different types of attacks mentioned above.

1.2 Project aim

The aim of this thesis is to propose methods of protecting multi-core processors against schedule-based attacks. This is achieved by introducing randomness in the multi-core execution schedule in such a way so that no deadline is met and the security of the system can be increased.

The work also included defining a metric to measure resiliency against different types of attacks. With this metric as baseline, task allocation and priority assignment algorithms were studied to understand their impact on security.

Several precautionary techniques were designed and evaluated to study how they protect against different types of attacks, and they proved to be highly effective. These include: Task migration, Reprioritizing and Pushback (forcing tasks to finish their job as late as possible).

1.2.1 Restrictions

Scheduling tasks on multi-core systems introduces several complications. Tasks can be assigned globally (able to migrate freely between processors) or partitioned (statically assigned to specific cores). For global scheduling, an exact test proving a task set schedulable without first scheduling the task set for an entire hyperperiod does not yet exist [8]. This research considers only partitioned scheduling, due to the restricted time frame of the project.

The system model presented when simulating the system does not account for the overhead normally introduced by context switches and task migrations. Simulating such overhead requires more advanced calculations and analysis than what is used in this work.

1.2.2 Challenges

For this project, all multi-core scheduling will be done using partitioned scheduling, as it enables the well-established theory on uni-core systems to be used when making scheduling decisions. When taking a partitioned approach, finding a feasible assignment of the tasks to the processors is NP hard, making larger task sets harder to prove schedulable [6]. The task allocation problem is analogous to the bin-packing problem, and we will investigate different bin packing algorithms commonly used when allocating tasks. The approach used when allocating tasks dictates the final task assignment, and in turn which schedules can be generated. To develop a security-aware scheduling algorithm for multi-core systems, the task assignment process will have to be researched to understand its impact on the total system security.

The maximum total randomness of the schedule is heavily affected by how the tasks are assigned to the processors. When a task assignment results in a processor experiencing high utilization, its potential to randomize its schedule decreases as

there is less flexibility to delay high priority tasks in favor of executing lower priority tasks at unexpected times. Analyzing and understanding how the assignment affects the randomness for the entire system is key for achieving a high security solution.

The aim of this research is to study not only the task allocation and its implications on security, but also to employ online randomization techniques such as re-allocations and controlled task migrations. Further, when developing a metric for assessing the security of a multi-core system, the attacker model is of great importance, as randomization techniques developed without considering security may not necessarily be effective or could even lead to create easier ways to compromise security. Multiple different types of attacks were considered, as well as attackers with varying capabilities.

An attacker attempting to manipulate a piece of data about to be read by another task might only be able to do so if both tasks execute on the same core. In other cases, if these tasks share system-wide resources, an attacker might be able to attack across cores. The developed metrics will be used to determine how to perform task allocation in a security-aware manner and help fine-tune the scheduling algorithm by introducing controlled randomness.

1.3 Contributions

As of today, there is no known research being made about secure real-time scheduling for multi-core processors considering partitioned scheduling in the context of randomization of schedules, making this work a notable contribution to the field.

More specifically, this thesis will contribute the following:

- Introduce new metrics tailored at multi-core systems for measuring schedule entropy across cores in a partitioned setup.
- Introduce a new metric specifically aimed at identifying the resiliency against certain type of attack based on when tasks execute in relation to each other. This will help develop techniques to protect against attacks where a malicious task is able to read or manipulate data before or after it is used by a trusted task.
- Study the properties of traditional partitioning algorithms and priority assignments to determine how they affect security using the proposed metrics.
- Improve the efficiency of the previously developed TaskShuffler by using a more precise inversion budget computation which would enable more flexible randomization of the schedule without violating any deadline.
- Given a feasible partition of the tasks, we implement novel precautionary online attack protection techniques to further strengthen the security on single-core as well as multi-core systems in relation to the proposed metrics. These include: Reprioritizing, Migration and Pushback.
- These protection techniques were all tested separately and in combination with

each other to understand how to best protect a system against different types of attacks.

- All simulations were run using a custom simulation framework built in C, which will be available to use in future research.

1.4 Thesis outlook

This thesis is divided up in chapters:

Chapter 1: *Introduction* outlines the aim of this thesis and introduces the reader to previous work in the research area.

Chapter 2: *Theory* provides a solid foundation to the basics of real-time scheduling of periodic tasks. Later, the algorithms to be used in the thesis are thoroughly explained. Finally, this chapter contains definitions of the metrics used to score and compare different configurations and scheduling algorithms.

Chapter 3: *Method* states which modifications and parameters will be used in an attempt to further improve on existing scheduling methods.

Chapter 4: *Results* presents the outcome of the simulations detailed in the previous chapter and attempts to explain the findings.

Chapter 5: *Conclusion* summarizes the key takeaways form the results of the simulations and reasons about which modifications had the biggest impact on security, and why.

2

Theory

This section provides the theoretical foundation necessary to understand the scheduling techniques and threat models addressed in this thesis. We begin by introducing the fixed-priority rate-monotonic scheduler for uni-core systems, a widely studied and fundamental algorithm in real-time systems theory. These concepts are then extended to multi-core platforms using a partitioned scheduling approach, where tasks are statically assigned to specific cores. Since the algorithm developed in this thesis builds upon fixed-priority partitioned scheduling, a thorough understanding of these principles is essential. Additionally, we introduce the key metrics used to evaluate system behavior under adversarial conditions, which will be instrumental in later chapters when analyzing and optimizing system resilience against attacks.

2.1 Background

2.1.1 The Fixed-Priority Rate-Monotonic scheduler

A **task set** $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}$ is a collection of N **periodic tasks**. Each task τ_i is described by a tuple (C_i, D_i, T_i), where:

- C_i is the *worst-case execution time*, which is the maximum amount of time the task may take to complete.
- D_i is the *relative deadline*, which is the maximum time allowed from the task's release until its completion.
- T_i is the *period*, which is the time between successive releases of the task.

If $D_i = T_i$, the task has an *implicit deadline*, meaning it must complete before the next release. If $D_i < T_i$, the task has a *constrained deadline*.

The least common multiple of all periods in the taskset is known as the *hyperperiod*. Under a deterministic scheduler, the execution schedule will repeat itself every hyperperiod.

In **fixed-priority scheduling**, each task is assigned a fixed priority before execution starts. The priorities are based on static task parameters, such as deadlines or periods, and do not change during runtime.

The **Deadline-Monotonic (DM) scheduler** assigns higher priority to tasks with

shorter relative deadlines. DM is *optimal* among all preemptive fixed-priority schedulers for constrained deadlines. This means that if any other fixed-priority scheduler can schedule a given task set without missing deadlines, DM can also do so.

When all tasks have implicit deadlines ($D_i = T_i$ for all i), the **Rate-Monotonic (RM) scheduler** is typically used. RM assigns higher priorities to tasks with shorter periods. In this case, RM and DM give the same priority assignment, and RM is also optimal.

The scheduler is assumed to be **preemptive**, meaning a higher-priority task can interrupt a lower-priority one if it becomes ready while the lower-priority task is still running.

RM Scheduling Example

We consider the following task set under Rate-Monotonic (RM) scheduling:

$$\Gamma = \{\tau_1 = (C_1 = 1, T_1 = 4), \quad \tau_2 = (C_2 = 1, T_2 = 5), \quad \tau_3 = (C_3 = 3, T_3 = 8)\}$$

RM priority order is: $\tau_1 > \tau_2 > \tau_3$.

Below is the timeline from time 0 to 12:

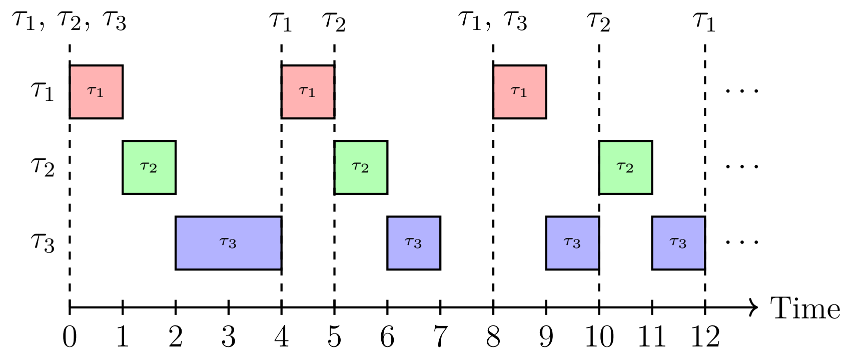


Figure 2.1: Example of an execution schedule generated by a fixed-priority scheduler.

In this schedule:

- At time 0, all tasks arrive. τ_1 runs first (highest priority), followed by τ_2, then τ_3.
- At time 4, τ_1 is released, preempting τ_3
- Likewise, at time 5, τ_2 is released, running before τ_3 is allowed to finish.
- At time 6, τ_3 is allowed to finish.
- At time 8, both τ_1 and τ_3 are released. τ_1 runs first (highest priority).
- At time 10, τ_2 is released, preempting τ_3, which resumes its execution at time 11.

2.1.2 Feasibility Tests

A **feasibility test** is used to check whether a given task set can be successfully scheduled by a specific scheduling algorithm, such as a fixed-priority scheduler. In other words, the test helps determine whether all tasks can meet their deadlines during execution.

There are three types of feasibility tests:

- **Necessary test**: If a task set fails this test, it is definitely not schedulable. However, passing the test does not guarantee that the task set is schedulable.
- **Sufficient test**: If the test is passed, the task set is guaranteed to be schedulable. But if it fails, we cannot be sure whether it is schedulable or not.
- **Exact test**: This type of test gives a definitive answer. The task set is schedulable if and only if the test passes. It is thus both necessary and sufficient.

Note that this exact test is more computationally expensive than simpler tests. It runs in **pseudo-polynomial time**, which can become costly for large task sets.

2.1.2.1 Response-Time Analysis (Exact Test)

To check if a task set is schedulable under fixed-priority scheduling, we can compute the *worst-case response time* for each task. This is the longest time from when a task is released to when it finishes, considering interference from higher-priority tasks.

For a task τ_i, its response time R_i is given by the recurrence:

$$R_i = C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \tag{2.1}$$

where:

- C_i is the execution time of τ_i.
- $\mathrm{hp}(\tau_i)$ is the set of tasks with higher priority than τ_i.
- T_j is the period of each higher-priority task.
- $R_i^0 = C_i$ and the recurrence is computed until $R_i^{k+1} = R_i^k$.

If $R_i \leq D_i$ for all tasks, then the task set is **schedulable**. This test is **exact**, meaning it provides a definitive answer.

Example: Response-Time Analysis under RM Scheduling

Consider the following task set:

$$\Gamma = \{\tau_1 = (C_1 = 1, T_1 = 4), \quad \tau_2 = (C_2 = 2, T_2 = 5), \quad \tau_3 = (C_3 = 3, T_3 = 10)\}$$

Here, implicit deadlines are assumed ($D_i = T_i$), so tasks with shorter periods receive higher priority:

$$\text{Priority order: } \tau_1 > \tau_2 > \tau_3$$

We now apply response-time analysis to each task, starting from the highest priority.

Step 1: τ_1 (highest priority) No interference from other tasks:

$$R_1 = C_1 = 1 \leq D_1 = 4 \quad \Rightarrow \text{OK}$$

Step 2: τ_2 (second highest priority) Interference from τ_1 only:

$$R_2^0 = C_2 = 2$$

$$R_2^1 = C_2 + \left\lceil \frac{R_2^0}{T_1} \right\rceil C_1 = 2 + \left\lceil \frac{2}{4} \right\rceil \cdot 1 = 2 + 1 = 3$$

$$R_2^2 = 2 + \left\lceil \frac{3}{4} \right\rceil \cdot 1 = 2 + 1 = 3 \quad \Rightarrow R_2 = 3 \leq D_2 = 5 \quad \text{OK}$$

Step 3: τ_3 (lowest priority) Interference from τ_1 and τ_2:

$$R_3^0 = C_3 = 3$$

$$R_3^1 = 3 + \left\lceil \frac{3}{4} \right\rceil \cdot 1 + \left\lceil \frac{3}{5} \right\rceil \cdot 2 = 3 + 1 + 2 = 6$$

$$R_3^2 = 3 + \left\lceil \frac{6}{4} \right\rceil \cdot 1 + \left\lceil \frac{6}{5} \right\rceil \cdot 2 = 3 + 2 + 2 = 7$$

$$R_3^3 = 3 + \left\lceil \frac{7}{4} \right\rceil \cdot 1 + \left\lceil \frac{7}{5} \right\rceil \cdot 2 = 3 + 2 + 4 = 9$$

$$R_3^5 = 3 + \left\lceil \frac{9}{4} \right\rceil \cdot 1 + \left\lceil \frac{9}{5} \right\rceil \cdot 2 = 3 + 3 + 4 = 10$$

$$R_3^6 = 3 + \left\lceil \frac{10}{4} \right\rceil \cdot 1 + \left\lceil \frac{10}{5} \right\rceil \cdot 2 = 3 + 3 + 4 = 10 \quad \Rightarrow R_3 = 10 \leq D_3 = 10 \quad \text{OK}$$

Conclusion: All tasks have response times less than or equal to their deadlines. Therefore, the task set is **schedulable under RM scheduling**.

2.1.2.2 Response-Time Analysis with Jitter

If tasks may not be released exactly on time (due to delays or jitter), we use an extended version of the response time formula [9]:

$$R_i = C_i + \sum_{\tau_j \in \text{hp}(\tau_i)} \left\lceil \frac{R_i^k + J_j}{T_j} \right\rceil C_j \tag{2.2}$$

The formula calculates the response time of task τ_i from the point of its release. Thus, the constraint for τ_i meeting all its deadlines becomes:

$$R_i + J_i \leq D_i$$

Here, J_j is the *release jitter* of task τ_j, representing the maximum delay from its expected release time. Jitter may occur because of scheduling overhead or a task awaiting the arrival of a message before released for execution.

This jitter-aware version will be used during the analysis of the scheduler developed later, as it better reflects real-world systems where jitter may occur.

2.1.2.3 Response-Time Analysis during execution

When proving feasibility for a taskset where tasks might be mid-execution, two tests must be applied. First, the priority ordering must be feasible under the ordinary response time test (2.1). Second, Each task must also be able to handle interference from higher-priority tasks during the remainder of its current period. Figure 2.2 illustrates this scenario, where the remaining execution of a high-priority task τ_k—delayed due to jitter or prior preemption—is handled separately from its future periodic invocations.

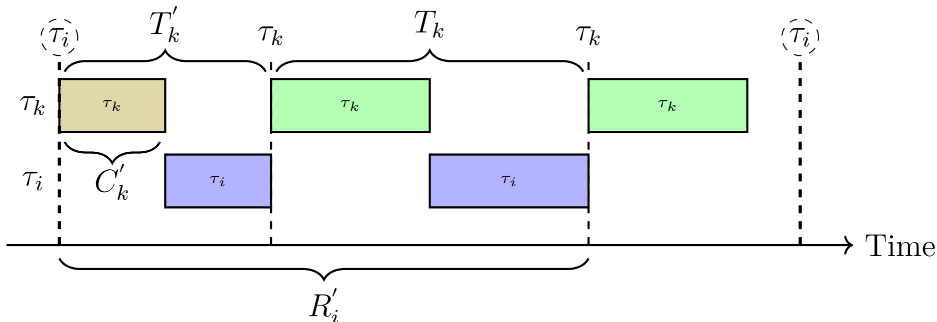


Figure 2.2: Response time R_i' of task τ_i after a potential migration or priority change, showing remaining execution of a high-priority task τ_k and its subsequent arrivals.

Consider the interference a task τ_i will experience for the remainder of its period after a sudden migration or priority assignment while task are already executing, as seen in figure 2.2. Based on the original response time equation 2.1, the specific response time for τ_i i given by:

$$R_i = C_i^{'} + J_i^{'} + \sum_{\tau_k \in \mathrm{hp}(\tau_i)} \left(C_k^{'} + \left\lceil \frac{R_i - T_k^{'}}{T_k} \right\rceil C_k \right) \tag{2.3}$$

where:

- $C_i^{'}$ is the remaining execution time of τ_i.
- $C_k^{'}$ is the remaining execution time of τ_k.
- C_k is the full execution time of τ_k.

- $T_k^{'}$ is the time until the next arrival of τ_k.
- T_k is the time until the next arrival of τ_i.
- $J_i^{'}$ is the remaining release jitter of τ_i, assumed maximal if τ_i has not yet started.

Initially, R_i^0 may be set to C_i'.

The formula considers a problem window smaller than the full response time by handling the first contribution of each high priority task separately. Naturally, τ_i must endure the interference caused by the current remaining execution time of a high priority task τ_k. The remaining interference is then calculated recursively but only considering tasks invoked after $T_k^{'}$ in to the window.

If the system approaches the boundary of a hyperperiod, this equation converges to the standard response time test (2.1).

If $R_i \leq D_i^{'}$ for all tasks, where $D_i^{'}$ is the remaining deadline of the current invocation of the task, the task set is feasible under the given priority ordering in its current state.

2.1.3 Optimal Priority Assignment (OPA)

The Rate-Monotonic (RM) scheduling policy assigns higher priorities to tasks with shorter periods. While RM is optimal when all tasks have implicit deadlines and release jitter is absent, it may fail to find a valid priority ordering when those assumptions do not hold, even though one exists.

To handle such cases, the **Optimal Priority Assignment (OPA)** algorithm, proposed by Audsley et al. [10], offers a more flexible approach. OPA, using a given schedulability test, can find a feasible fixed-priority assignment that guarantees all tasks meet their deadlines, if such an assignment exists.

How OPA works:

OPA assigns task priorities in a bottom-up order, starting from the *lowest* available priority and working its way up. At each step:

1. OPA considers all tasks that have not yet been assigned a priority.
2. It tests each one to see if it would still meet its deadline if it were given the current lowest priority, assuming the remaining tasks have higher priorities.
3. If such a task is found, it is permanently assigned that priority and removed from the unassigned list.
4. The process repeats with the remaining tasks until all have been assigned a priority.

If no task is schedulable at a given step, OPA reports failure. This means there is no fixed-priority assignment using the same test as used during OPA that can make the task set schedulable.

The key idea is that a task's schedulability depends only on which tasks have higher priority, not the exact order among them. This allows OPA to test candidates independently and still ensure optimal results.

Algorithm 1 Optimal Priority Assignment (OPA)

```
1:  Γ ← set of all tasks
2:  for p = |Γ| to 1 do                                  ▷ Start from lowest priority
3:      for each task τ ∈ Γ do
4:          Temporarily assign priority p to τ
5:          Temporarily assign higher priorities to tasks in Γ \ {τ}
6:          if τ is schedulable using a test S then
7:              Assign fixed priority p to τ
8:              Remove τ from Γ
9:              break                                    ▷ Move to next priority level
10:         end if
11:     end for
12:     if no task was schedulable using a test S then
13:         return Failure
14:     end if
15: end for
```

OPA is **provably optimal** among all fixed-priority assignments: if it is possible to schedule a task set using any fixed-priority order, OPA will find it. However, this guarantee depends on the schedulability test used.

In this thesis, we always use **response-time analysis (RTA)** as our schedulability test. Since RTA is exact, it guarantees that OPA will succeed if and only if a feasible fixed-priority assignment exists.

OPA is useful in cases when scheduling under a certain priority heuristic, such as Increasing Utilization (IU), is impossible, but we still want to find a feasible priority assignment following this ordering as much as possible. In such cases, a variation of OPA can be used. Instead of aborting the search of current priority level as soon as a feasible task is found (line 6-9), OPA can search for and store all tasks able to be assigned to the current priority level. The algorithm then picks the task according to a desired heuristic. To find a feasible priority assignment resembling IU, the algorithm saves the low utilization tasks for the high priority spots by picking the highest utilization task to run at as low priority levels as possible.

In the chapters that follow, OPA will be a key component of the scheduling framework developed in this thesis. It will help ensure that randomly generated or obfuscated task schedules still meet all deadlines, even in the presence of jitter.

2.1.4 Partitioned Scheduling in Multi-core Systems

In systems with multiple processors, one common approach to scheduling is called *partitioned multi-core scheduling.* In this model, each task in the system is assigned to exactly one processor. Once assigned, that processor is responsible for scheduling its tasks independently, often using a uni-core scheduling algorithm such as Rate-Monotonic (RM) or Deadline-Monotonic (DM).

This setup transforms the overall scheduling problem into two parts:

1. **Partitioning** the task set across the available processors.
2. **Scheduling** each group of tasks on its processor.

Partitioning is typically done using a class of methods known as *bin-packing algorithms* [6]. In this context, each processor is treated like a bin, and tasks are the items to be placed. The goal is to distribute tasks across processors in such a way that each processor's set of tasks is schedulable.

Because task scheduling depends on how tasks interact with each other (for instance, through shared deadlines or overlapping execution times), simply placing tasks without considering their timing constraints is not sufficient. Every time a task is assigned to a processor, a *feasibility test* must be run to check whether all tasks on that processor will still meet their deadlines after the new task is added. This test might involve calculating worst-case response times or checking processor utilization.

Partitioning the task set is a difficult problem: in fact, it is **NP-hard** [6]. This means there is no known method that can quickly find the optimal assignment in all cases without checking a vast number of combinations. As a result, practical systems use heuristic algorithms that try to find a "good enough" partition by making smart decisions based on certain rules or priorities.

Key Factors in Partitioning Algorithms

Bin-packing-based partitioning algorithms usually rely on two main decisions:

1. The order in which tasks are considered for assignment.
2. The method used to choose which processor a task should be assigned to.

Task Ordering Strategies

The order in which tasks are handled can have a major impact on the resulting partition. Common strategies include:

- **Rate-Monotonic (RM)**: Tasks are sorted by increasing period. Tasks with shorter periods are assigned first.
- **Deadline-Monotonic (DM)**: Tasks are sorted by increasing deadline. Tasks with tighter deadlines are given priority.
- **Slack-Monotonic (SM)**: Tasks are ordered by increasing slack time (deadline minus execution time). Tasks with less slack are assigned first.

- **Increasing Utilization (IU)**: Tasks are sorted by increasing utilization. Tasks that use fewer resources are placed first.
- **Decreasing Utilization (DU)**: Tasks are sorted by decreasing utilization. Tasks that are more demanding are placed first.

Processor Assignment Strategies

Once the task order is decided, the next step is to assign each task to a processor. This is where bin-packing strategies come into play. Popular strategies include:

- **First-Fit (FF)**: The task is placed on the first processor where it fits (i.e., passes the feasibility test).
- **Best-Fit (BF)**: The task is placed on the processor where it leaves the least remaining space (e.g., lowest unused utilization) without violating feasibility.
- **Worst-Fit (WF)**: The task is placed on the processor with the most available capacity, spreading the workload more evenly.

Each combination of task ordering and processor assignment strategy can produce different results, both in terms of schedulability and how evenly tasks are distributed across processors.

In this thesis, we explore how different partitioning strategies affect not only schedulability but also system-level properties such as entropy and resilience. Partitioned scheduling provides a structured way to manage task allocation in multi-core systems and serves as a foundation for more advanced scheduling techniques presented in later chapters.

2.2 TaskShuffler

TaskShuffler is a real-time scheduling algorithm based on the well-known fixed-priority Rate-Monotonic (RM) scheduler [5]. Unlike the traditional RM scheduler, which always selects the highest-priority ready task to run, TaskShuffler introduces controlled randomness into the schedule. This is done by allowing lower-priority tasks to occasionally delay higher-priority ones, as long as the deadlines of higher-priority tasks are still met. This is visualized in figure 2.3 and 2.4, where the use of TaskShuffler makes the schedule unpredictable, obfuscating the execution schedule across hyperperiods, while still ensuring tasks complete execution before their deadline.

Inversion Budgets

To manage this behavior safely, TaskShuffler calculates a value called the *Worst-case Maximum Inversion Budget* for each task. This value defines how much time a task can be delayed by lower-priority tasks without missing its deadline. The inversion budget for task τ_i is calculated as:

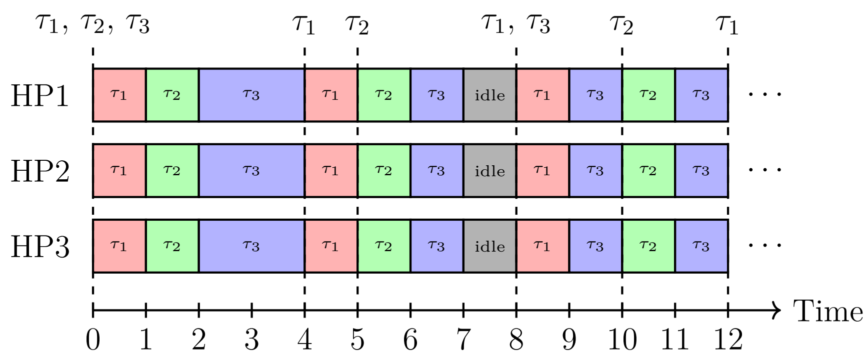


Figure 2.3: Execution schedule across several hyperperiods without TaskShuffler.

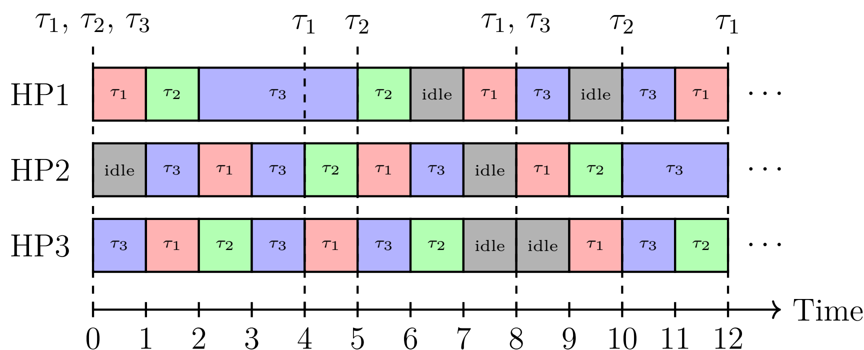


Figure 2.4: Shuffled execution of tasks across three hyperperiods using TaskShuffler.

$$V_i = D_i - \left[C_i + J_i + \sum_{\tau_j \in \text{hp}(\tau_i)} \left(\left\lceil \frac{D_i}{T_j} \right\rceil + 1 \right) C_j \right] \tag{2.4}$$

Here, D_i is the deadline of task τ_i, C_i is its worst-case execution time, J_i is the maximum release jitter of the task, T_j is the period of a higher-priority task τ_j and $\text{hp}(\tau_i)$ is the set of all tasks with higher priority than τ_i.

Each task's inversion budget is tracked at runtime. When a lower-priority task executes while a higher-priority task still has work to do, the budget of the higher-priority task is reduced. At the beginning of each period, a task's inversion budget is reset to its original value.

Making Scheduling Decisions

At every scheduling decision point, TaskShuffler reviews the set of ready tasks. These tasks are examined in decreasing order of priority, beginning with the highest-priority task τ_1. For each task τ_i in the ready queue, the algorithm checks its current inversion budget V_i:

- If $V_i > 0$, then the algorithm considers letting a lower-priority task execute temporarily.
- If $V_i \leq 0$, the task cannot be delayed further.

TaskShuffler also calculates a threshold priority called the *minimum inversion priority* M_i for the highest-priority task in the ready queue. The purpose of M_i is to not let any low priority task execute ahead of a high priority task with positive budget such that this high priority task suffers release jitter and cause extra interference on another (possibly yet to come) low priority task with no budget. This value is defined as:

$$M_i = \max\left(\mathrm{pri}(\tau_j) \mid \tau_j \in \mathrm{lp}(\tau_i) \text{ and } V_j < 0\right) \tag{2.5}$$

If no such task exists, M_i is set to an arbitrarily low priority. The set of candidate tasks includes all ready tasks with priority greater than or equal to M_1. If a task is encountered with an inversion budget of zero or less, the algorithm stops adding more candidates.

Randomized Execution

From the list of candidate tasks, one is selected at random to execute. The algorithm also chooses a random execution time for the selected task, bounded by the smallest remaining inversion budget among all the higher-priority tasks. The selected task runs for either this duration or until a new task arrives, whichever happens first. At that point, a new scheduling decision is made, and the process repeats.

Including the Idle Task

To give TaskShuffler more flexibility, an idle task is added to the task set. This idle task has an infinite period and execution time, representing processor inactivity. Including this task allows the scheduler to consume inversion budgets fully, making task execution less predictable and more randomized. Without the idle task, the scheduler would be forced to execute real tasks immediately, reducing the amount of randomness in the schedule.

2.2.1 Measuring Randomness with Schedule Entropy

In order to evaluate how unpredictable a task schedule is, we need a metric that can quantify its randomness. One such metric is called *schedule entropy*, which was proposed in TaskShuffler [5]. Schedule entropy captures the variability of execution patterns across multiple repetitions of a schedule.

Concept of Schedule Entropy

Suppose we execute the same task set multiple times over its hyperperiod (the least common multiple of the task periods), and record which task is scheduled at each time slot. If the execution pattern is always the same, the schedule is highly predictable and its entropy is low. On the other hand, if the pattern changes frequently between runs, the schedule is more unpredictable and the entropy is higher.

Formally, consider a set S of observed schedules over multiple hyperperiods. Each schedule has a fixed length L (the number of time slots in one hyperperiod), and each slot in the schedule contains the identity of the task that was running at that time.

The *schedule entropy* $H_\Gamma(S)$ is defined as the Shannon entropy over the probability distribution of complete schedules:

$$H_\Gamma(S) = -\sum_{s_0=1}^{N} \cdots \sum_{s_{L-1}=1}^{N} \Pr(s_0, \ldots, s_{L-1}) \log_2 \Pr(s_0, \ldots, s_{L-1})$$

Here:

- s_t is the task scheduled at time slot t, where $s_t = i$ if task τ_i is scheduled.
- N is the total number of tasks, including the idle task.
- $\Pr(s_0, \ldots, s_{L-1})$ is the probability that a specific sequence of task executions occurs over one hyperperiod.

By convention, any term where $\Pr(s_0, \ldots, s_{L-1}) = 0$ contributes zero to the entropy sum.

Approximation Using Slot Entropy

Computing the full schedule entropy is infeasible in practice, since the number of possible schedules grows exponentially with the schedule length L. Specifically, there are $O(N^L)$ possible combinations of task sequences.

To address this, we can approximate the total entropy by calculating the entropy at each individual time slot and summing these values. This is known as the *slot entropy*:

$$H_\Gamma(S_t) = -\sum_{s_t=1}^{N} \Pr(s_t) \log_2 \Pr(s_t) \tag{2.6}$$

where $\Pr(s_t)$ is the probability that task τ_i appears at slot t across all sampled schedules.

The approximate schedule entropy $\widetilde{H_\Gamma}(S)$ is then computed by summing the slot entropies:

$$\widetilde{H_\Gamma}(S) = \sum_{t=0}^{L-1} H_\Gamma(S_t) \tag{2.7}$$

This sum provides an upper bound on the true schedule entropy:

$$H_\Gamma(s_0, \ldots, s_{L-1}) \leq H_\Gamma(s_0) + \cdots + H_\Gamma(s_{L-1})$$

Equality holds only when the scheduling decisions at each slot are completely independent of one another, which is generally not the case in real-time systems. Since what happens at one slot affects future decisions, the true entropy is lower than the sum of individual entropies.

Despite this, the approximation $\widetilde{H}_\Gamma(S)$ has been shown to correlate well with the actual schedule entropy [5], and is commonly used as a practical substitute. For simplicity, we will refer to this approximation as $H_\Gamma(S)$ in the remainder of this work.

2.2.2 Improving taskshuffler

The inversion budget calculation (2.4) used in TaskShuffler is pessimistic, as it adds an entire extra full-length duration of a high-priority task when considering a back-to-back hit. This can be refined using a more precise interference model, illustrated in Figure 2.5.

In the worst case, a high-priority task τ_k released prior to τ_i may be delayed such that it executes as late as its budget allows. This occurs when τ_k experiences maximum jitter and has already consumed its full inversion budget. The number of full executions of τ_k within the extended problem window $D_i + V_k + J_k$ is:

$$N_k = \left\lfloor \frac{D_i + V_k + J_k}{T_k} \right\rfloor$$

The remaining portion of the window, which may only partially accommodate another execution of τ_k, is:

$$\epsilon = D_i + V_k + J_k - N_k \cdot T_k$$

The interference caused in this partial window is limited to $\min(C_k, \epsilon)$, giving a total interference from τ_k:

$$I_k = N_k \cdot C_k + \min(C_k, \epsilon)$$

Using this more precise calculation, the maximum inversion budget for task τ_i becomes:

$$V_i = D_i - C_i - J_i - \sum_{\tau_k \in \mathrm{hp}(\tau_i)} I_k$$

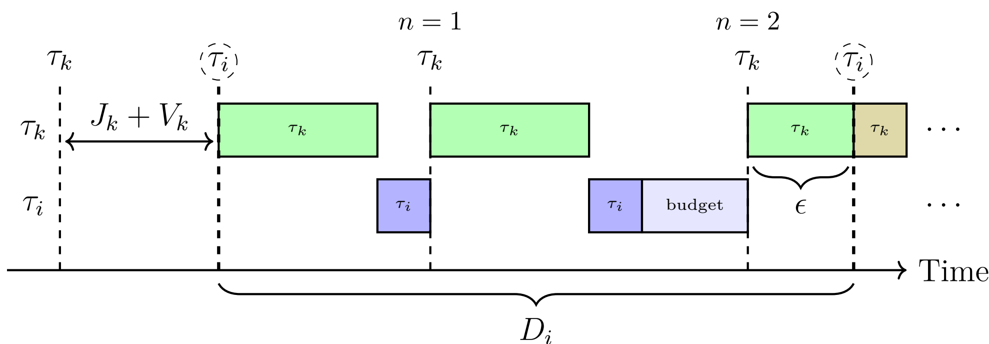


Figure 2.5: Illustration of high-priority task interference within $D_i + V_k + J_k$.

The impact of using this tighter, less pessimistic inversion budget will be evaluated in Section 4.

2.3 Metrics

In this section, we extend the concept of schedule entropy from single-core systems to multi-core systems. We also introduce a security-aware metric that helps evaluate how easily a trusted task might be targeted by an attacker.

We consider two main scenarios for attackers in multi-core systems:

- **Horizontal case:** The attacker can observe the execution of each individual core.
- **Vertical case:** The attacker sees which tasks are running at each time, but does not know which core they are running on.

2.3.1 Multi-core Schedule Entropy

Each processor core has its own schedule, and a low-entropy (i.e., predictable) schedule on just one core could create a security vulnerability. To generalize schedule entropy to multi-core systems, we define two types of multi-core entropy: horizontal and vertical.

Horizontal Multi-core Entropy

In the horizontal case, the attacker can only monitor or attack the core it is currently running on. Therefore, we must ensure that each core's schedule is independently unpredictable. We define the *horizontal multi-core schedule entropy* H^h_Γ as the geometric mean of the schedule entropies from all m cores:

$$H^h_\Gamma = \left(\prod_{i=1}^{m} H_\Gamma(S_i) \right)^{1/m} \tag{2.8}$$

where $H_\Gamma(S_i)$ is the schedule entropy of core i and m is the number of non-empty cores. The schedule entropy of an empty core is always zero, and including it would immediately zero the metric.

This formula ensures that if even one core has a very low entropy (i.e., a highly predictable schedule), the overall metric will also be low. Thus, for the system to be secure in this model, all cores must maintain high entropy.

Vertical Multi-core Entropy

In the vertical case, the attacker does not know which core a task is running on, but can observe which tasks are active at each point in time. This might happen if the attacker controls an external shared resource (e.g., memory) used by tasks from different cores. The goal is to measure how much the execution across cores appears correlated over time.

To calculate vertical entropy, we first distinguish the idle task on each core. Let $\tau_i^\emptyset$ represent the idle task on core i. At every time slot, we record the probability of each task executing on *any* core (including idle tasks). Since each of the m cores is always running some task (active or idle), the probabilities at any given time slot sum to m. Importantly, each task can only be active on one core at a time, so its maximum probability at any slot is 1.

We define the *vertical time slot entropy* at time slot t as:

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

where N is the total number of tasks (including all idle tasks), and $\Pr(s_t)$ is the probability that task s_t is scheduled at time t on *any* core.

The *vertical multi-core schedule entropy* is then defined as the average of the vertical slot entropies:

$$\widetilde{H}_\Gamma^v(S) = \frac{1}{m}\sum_{t=0}^{L-1} H_\Gamma(S_t) \tag{2.9}$$

This metric does not consider which core a task is scheduled on, only whether it is running at a particular time. It helps identify how predictable the overall system appears from an external point of view.

Example: Calculating Vertical Multi-core Entropy

To illustrate how vertical schedule entropy is computed, consider the following example. We calculate the time slot entropy at $t = t_0$ for a system with $m = 4$ cores and $n = 6$ tasks. Table 2.1 shows which task was executing on each core at time t_0 across the first four hyperperiods.

	HP1	**HP2**	**HP3**	**HP4**	...
m_1	$\tau_1^\emptyset$	τ_1	τ_3	τ_3	...
m_2	τ_4	τ_6	τ_5	τ_5	...
m_3	τ_2	τ_2	$\tau_3^\emptyset$	$\tau_3^\emptyset$	...
m_4	$\tau_4^\emptyset$	$\tau_4^\emptyset$	τ_2	τ_2	...

Table 2.1: Task assignments across four hyperperiods at time t_0 for a 4-core system. Empty slots are represented using the $\tau_i^\emptyset$ notation.

Using Table 2.1, we can estimate the probability P_i of each task τ_i executing at time t_0 by counting how many times it appears across the hyperperiods. These empirical probabilities are then used to compute the vertical schedule entropy using Equation 2.9. The complete statistics are presented in Table 2.2.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	$\tau_1^\emptyset$	$\tau_2^\emptyset$	$\tau_3^\emptyset$	$\tau_4^\emptyset$	$\sum$
$\#_i$	1	4	2	1	2	1	1	0	2	2	16
P_i	25%	100%	50%	25%	50%	25%	25%	0%	50%	50%	400%
$-H_\Gamma(S_t)_i$	0.5	0	0.5	0.5	0.5	0.5	0.5	0	0.5	0.5	4

Table 2.2: Empirical frequencies ($\#_i$), estimated probabilities (P_i), and contribution to time slot entropy ($-H_\Gamma(S_t)_i$) for each task at $t = t_0$.

The total number of execution slots across 4 hyperperiods is $4 \cdot m = 16$. Given that all cores are active during each hyperperiod, the system utilization is 400% ($m = 4$). The total time slot entropy at $t = t_0$ is 4, resulting in a vertical schedule entropy contribution of $4/m = 1$.

It is important to note that estimating schedule entropy using data from only a few hyperperiods may not accurately reflect the true probability distribution of task execution at each time slot. This example is intended solely for illustrative purposes.

Example: Optimal Vertical Schedule Entropy

To understand the upper bound of vertical schedule entropy, consider the case where the distribution of task execution is perfectly uniform. That is, at time $t = t_0$, every possible task (including idle tasks) has an equal probability of being scheduled across the system.

Let the system have m cores and n real tasks. We also include m idle "tasks" to represent the possibility of cores being idle, resulting in $n + m$ total possible task identifiers at any given time slot. In this optimal entropy scenario, each task (including idle tasks) appears with equal probability:

$$P_i = \frac{m}{n+m} \quad \forall i \in \{1, \ldots, n+m\}$$

We now compute the vertical schedule entropy using Equation 2.9. Since all P_i are equal, we can simplify the entropy expression significantly.

$$\begin{aligned} H_\Gamma(S_t) &= \frac{1}{m}\sum_i -P_i \cdot \log_2(P_i) \\ &= \frac{1}{m} \cdot (n+m) \cdot \left(-\frac{m}{n+m} \cdot \log_2\left(\frac{m}{n+m}\right)\right) \\ &= -\frac{1}{m} \cdot m \cdot \log_2\left(\frac{m}{n+m}\right) \\ &= -\log_2\left(\frac{m}{n+m}\right) \\ &= \log_2\left(\frac{n+m}{m}\right) \\ &= \log_2(n+m) - \log_2(m) \end{aligned} \tag{2.10}$$

Key takeaways.

- The resulting entropy is highest when the number of tasks is large relative to the number of cores. If the system expands using more cores, more tasks would theoretically need to be added to maintain the same entropy score.
- The expression $\log_2(n+m) - \log_2(m)$ quantifies the maximum uncertainty the scheduler can exhibit at any time slot, normalized by the number of cores. By distinguishing between “real” and “idle” tasks, we are able to show that normalizing the formula by the number of cores ensures that the system doesn’t appear safe just because the execution is spread out across multiple cores.

2.3.2 Security-Aware Metrics

As discussed in [11], schedule entropy is not inherently security-aware. It measures how disordered or random the schedule appears over different hyperperiods but does not consider the relationship between specific tasks. For example, it does not capture how often a potentially malicious task executes immediately before a sensitive or trusted task.

To evaluate security more directly, we introduce a metric that analyzes how predictable the execution order of tasks is, particularly when considering task interactions that could lead to security vulnerabilities.

Anterior, Posterior and Pincer attacks

As mentioned in [11], common schedule-based attacks include:

- **Anterior** Attacker task executes after the release of the victim task but before the time slot if first executes on the processor.
- **Posterior** Attacker executes after the completion of the victim task, but before the end of its period

- **Pincer** Attacker perform both an anterior and posterior attack.

The metric proposed in this section may be used to quantify how well the system is protected against all these kinds of attacks, merely by defining f to be the probability of the considered attack to occur.

Task Interaction Probability

Given a task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, let the probability function

$$f : T \times T \rightarrow [0, 1]$$

represent the probability of the task interaction of interest. Specifically, $f(\tau_i, \tau_j)$ is the probability that task τ_i performs an attack on τ_j. In the case of an anterior attack, $f(\tau_i, \tau_j)$ would represent the probability that τ_i (attacker) executes after the release of τ_j (victim) but before the time slot task τ_j first executes on the processor.

For a given task $\tau_j \in \Gamma$, we define its *maximum vulnerability* as:

$$d(\tau_j) = \max_{\tau_i \in \Gamma, \tau_i \neq \tau_j} f(\tau_i, \tau_j)$$

This captures the greatest risk to τ_j, assuming the worst-case scenario where the same task τ_i always precedes it.

Attack resiliency metric

To measure the overall security of the system, later referred to as the *attack resiliency*, we compute the minimum remaining uncertainty across all tasks:

$$\min_{\tau_j \in \Gamma}(1 - d(\tau_j)) \tag{2.11}$$

This value expresses the lowest level of unpredictability in task ordering. If one task is consistently preceded by another, this metric will be close to zero, indicating a security risk.

However, this approach may be too conservative. In practice, not all tasks are equally sensitive, and not all tasks are potential attackers. Therefore, we refine the analysis by making reasonable assumptions about which tasks are trusted and which are not. Such assumptions may be informed by established methods like Threat Analysis and Risk Assessment (TARA), or derived from system properties such as mixed-criticality. For example, tasks developed by trusted vendors may be treated as non-malicious, whereas those from less-trusted sources could be considered potential adversaries.

Trusted vs. Untrusted Tasks

To improve the relevance of this metric, we divide the task set into two subsets:

- Γ_T: trusted tasks that must be protected
- Γ_U: untrusted tasks that could potentially act as attackers

We then evaluate the probability that any untrusted task performs the attack of interest on a trusted task:

$$d_{\text{trust}}(\tau_j) = \max_{\tau_i \in \Gamma_U} f(\tau_i, \tau_j), \quad \text{for } \tau_j \in \Gamma_T$$

and define the security metric over trusted tasks as:

$$\min_{\tau_j \in \Gamma_T} (1 - d_{\text{trust}}(\tau_j))$$

This version focuses only on scenarios where an untrusted task could influence a trusted one, providing a more realistic estimate of actual attack risk. This is the metric which will be used to measure resiliency against the attacks described above: anterior, posterior and pincer.

Horizontal vs. Vertical Analysis

Just like schedule entropy, this attack resiliency metric can be applied in two different ways:

- **Horizontal analysis:** considers only interactions between tasks running on the same core.
- **Vertical analysis:** considers all task interactions, regardless of which core they run on.

Horizontal analysis models threats like cache-based side-channel attacks where the attacker shares hardware resources with the victim. Vertical analysis represents attackers who observe global scheduling behavior but cannot identify the specific core a task runs on.

Together, these variants help evaluate different threat models and system designs.

2.3.3 Terminology

To clarify the usage of terminology described in this section:

- When measuring protection against a specific attack, eg. anterior, horizontal/vertical anterior security refers to the resiliency against this type of attack.
- When considering a uni-core processor, vertical and horizontal schedule entropy yields identical results, and they may both be referred to as just schedule entropy. Likewise, as horizontal and vertical attack resiliency is also identical on uni-core processors, the term attack resiliency may be used.

2.4 Task Set Generation

To evaluate scheduling algorithms and randomness metrics, we must generate a wide variety of synthetic task sets. These task sets should span different utilization levels and task configurations to reflect the diversity of real-world systems.

Generating Task Sets for Single-Core Systems

For uni-core processors, one of the most commonly used methods for generating task sets is the **UUniFast** algorithm [12]. UUniFast was introduced to generate task utilizations that:

- Appear uniformly distributed,
- Always sum to a specified total utilization U, where $0 \leq U \leq 1$.

This method is widely adopted in real-time systems research [13]–[17]. A key reason for its popularity is that it allows researchers to easily explore the performance of scheduling strategies across different system loads.

However, UUniFast is only suitable when the total utilization U does not exceed 1. If we attempt to generate a task set with $U > 1$ for a multi-core processor (where total utilization can exceed 1), the algorithm may generate individual tasks with utilization greater than 1. Such tasks are not feasible, as no processor can handle more than 100% of a single task's execution time within its period. UUnifast-Discard was proposed to solve this which throws away task set of a certain task's utilization in the set is higher than 100% and then retry. Since the number of retries may grow when the utilization is large, we apply a Stafford's RandFixedSum [18] algorithm.

Generating Task Sets for Multi-Core Systems

To generate feasible and realistic task sets for multi-core systems, a more flexible algorithm is needed. **Stafford's RandFixedSum algorithm** [18] addresses this need by:

- Generating a fixed number of values (task utilizations),
- Ensuring their sum equals a given total utilization U,
- Allowing for constraints on each individual utilization (e.g., values must be between 0 and 1).

This approach makes it possible to create valid task sets even when the total utilization exceeds 1, by ensuring each task still has a utilization below or equal to the core limit (typically 1.0). The generated tasks can then be assigned across multiple cores using partitioning scheduling methods.

Importance of Bounded Task Utilizations

In real-time systems, every task must be schedulable within its period. If a task's utilization exceeds 1, it means it would require more processing time than available within its own period, making it impossible to meet its deadlines. Therefore, bounding each task's utilization is crucial when generating task sets, particularly for multi-core scenarios.

The RandFixedSum algorithm provides this control, making it the preferred choice for simulation studies and evaluations involving multi-core scheduling strategies.

3

Methods

In this chapter, the structure of the research is presented in detail. Initially, the attacker model is defined, serving as basis for evaluating scheduling algorithms.

3.1 System and Threat Model

This section outlines the capabilities of the simulated system as well as the attacker. (Same task model as [5], [15], [7])

3.2 System Model

This thesis considers a real-time system consisting of a set of independent periodic tasks. Each task is a program that repeatedly performs some computation at regular time intervals. The system runs on a multi-core platform, where each core executes one task at a time, and tasks are assigned to specific cores in advance.

A periodic task τ_i is described by the tuple (T_i, C_i), where:

- T_i is the period, meaning the time between two consecutive activations of the task.
- C_i is the worst-case execution time, which is the maximum amount of processor time the task needs each period.
- The relative deadline, D_i, is equal to the task period. (i.e, implicit deadline)

Tasks release a new job upon each invocation and may experience release jitter up to 10% of their period. Since $D_i = T_i$, and assuming schedulability, each task can have at most one active job at any given time. For notational simplicity, we use τ_i to denote both the task and its jobs, using the terms interchangeably.

The hyperperiod L is defined as the least common multiple (LCM) of all task periods. The utilization of a task τ_i is given by $u_i = C_i/T_i$, and the total system utilization is $U = \sum_{i=1}^{N} u_i$.

Within each core, tasks are scheduled using fixed priorities, where higher-priority tasks can interrupt (preempt) lower-priority tasks. For a given task τ_i, we define $\mathrm{hp}(\tau_i)$ and $\mathrm{lp}(\tau_i)$ as the sets of tasks with higher and lower priorities than τ_i on the same core, respectively.

In line with prior work such as TaskShuffler [5] and Krüger et al. [7], we assume that tasks are independent and do not have any precedence constraints.

- Tasks are partitioned across cores; task migration may only occur if feasibility tests allow.
- No synchronization primitives are used; task interactions, if any, occur via shared memory.
- All task parameters are known and static.

As described in [11], a logical execution time (LET) paradigm is assumed. This restricts system interactions with the I/O devices to occur only at certain time instances such as task releases/deadlines, which in turn enables us to more clearly define when the system is vulnerable.

The input to a task may only be externally altered between its release and the first time unit of execution on the processor. Likewise, its output may only be externally altered between the end of execution and the task's deadline.

This simplified model captures the core behavior of many real-world embedded systems and provides a foundation for analyzing more complex scheduling strategies and security concerns in the following chapters.

3.2.1 Threat Model

This thesis considers a *defense-aware adversary*, meaning an attacker who understands how the system and its security mechanisms work. The attacker uses this knowledge to exploit the predictability of fixed-priority real-time schedules.

Following Kerckhoff's principle [19], we assume the attacker has full knowledge of the system. This includes the task set, scheduling algorithm, task priorities, and timing parameters. The attacker is also assumed to have access to an identical version of the system for testing and reverse engineering.

The attacker has compromised one or more tasks in the system. This could happen through software vulnerabilities or the inclusion of untrusted third-party code. The capabilities of the attacker are summarized as follows:

- Full knowledge of system configuration, including all task parameters and their priorities.
- Can fully control the compromised task during its time executing on the processor.
- Has access to shared memory or memory-mapped buffers to monitor or manipulate data,
- No ability to change the scheduler, task configuration, or move tasks between processor cores.

This represents a strong insider threat, which has been examined in prior work [11]. The attacker attempts to carry out *schedule-based attacks* by aligning its activity with

the execution of a targeted task (referred to as the victim). The goal is to observe or influence the victim's behavior through timing and shared resource interference.

We consider the following types of attacks:

- **Anterior:** The attacker executes immediately before the victim task. This allows it to prepare shared resources, such as the cache, or manipulate input data before the victim reads it.
- **Posterior:** The attacker executes right after the victim finishes. This allows it to tamper with the victim's output by modifying memory after the victim has written its result.
- **Pincer:** The attacker executes both before and after the victim. This gives the attacker the opportunity to monitor or alter data before and after the victim runs, enabling more complex forms of interference or observation.
- **Concurrent:** The attacker runs at the same time as the victim task on a different core. By using shared resources, such as system-wide caches, the attacker can infer information about the victim's execution patterns.

We consider two variations of the attacker:

- An attacker that can interact with tasks on any core by leveraging shared hardware resources such as caches.
- A more limited attacker that can only interact with other tasks running on the same core.

This threat model captures a realistic but challenging scenario, which motivates the development of stronger defenses against timing-based and schedule-aware attacks.

3.3 Protection strategies

Several mitigation strategies to prevent attacks are discussed below

3.3.1 TaskShuffler

As described in 2.2, the TaskShuffler algorithm randomizes task execution in an attempt to increase schedule entropy, mitigating concurrent attacks. The aim of this mitigation strategy is to obfuscate the execution schedule over time to decrease the predictability of the system, thus increasing schedule entropy.

3.3.2 Migration

To further increase local entropy, tasks may be able to migrate to other cores. When adding a task to a core during execution extra caution must be taken to ensure all future jobs will meet their deadlines. Thus, migration feasibility uses a custom version of the response time test. When considering a task migration, it is first given a preliminary priority.

For it to migrate, two response time tests must hold for every task τ_i:

- The ordinary response time test, Eq 2.1
- The special case response time test for tasks in the middle of execution, as is given in Eq. 2.3

This ensures that every task will be able to meet both their current and all upcoming deadlines.

This is an expensive test, as it requires performing two response time tests on all low-priority tasks at and below the changed priority ordering. To combat this, one could compute several feasible partitions and priority orderings ahead of time, randomly switching between these during execution to simulate online migration. However, migration mid-execution is difficult to test and analyze ahead of time, as it is highly dependent on the exact parameters of the tasks at the point of migration. At the hyperperiod boundary, all tasks are reset to a predictable state, making the use of precomputed partitions much more feasible.

In this setup, **migration** is implemented as follows:

- The feature is invoked every hyperperiod.
- When migration is invoked, a random task in the system is picked. Next, a random core is chosen for the task to migrate to.
- Both the ordinary and the special case response time test is perform to assess if the task is allowed to migrate. If both tests pass, the task is migrated.

3.3.3 Reprioritizing

If the priority ordering on a core is static, high priority tasks will tend to perform anterior attacks on low priority tasks. To circumvent this pattern, one can regularly reassign task priorities to alternate which tasks tend to be scheduled first.

When taskshuffler is used, tasks are picked randomly anyways. However, for high utilization tasksets, the available inversion budgets might be too low to allow for variation.

3.3.4 Pushback

To mitigate posterior attacks, no other task should be able to execute immediately after the completion of a victim task. To achieve this, the following modifications are made to TaskShuffler, pushing back the final execution of tasks as late as possible.

When picking from eligible tasks, avoid picking tasks with only 1 time unit of execution left as long as their budgets allow, as this would otherwise finish the task early.

If the task selected for execution is scheduled to execute until completion and inversion budget left, the time until the next scheduling decision is decremented by

one, making sure the task doesn't run until completion. If the remaining execution is equal to one, the task must run, as this would mean its budget is consumed.

3.4 Simulations

This section is dedicated to presenting details concerning how the simulations were structured.

3.4.1 Task set generation

Task sets are generated similarly to how it is done in TaskShuffler [5]. As depicted in Algorithm 2, given a hyperperiod H, number of tasks n, and total utilization U, a task set is generated as follows:

The hyperperiod H is fixed in advance. We construct a list of potential task periods by selecting divisors of H that are greater than a small threshold (e.g., 10) to avoid unrealistically short task periods. Each task is then assigned a random execution time, and its period is chosen from the candidate set to match the desired utilization as closely as possible. This ensures that the generated task set is consistent with the specified total utilization.

Algorithm 2 Generate Task Set with Total Utilization U and Hyperperiod H

Require: Number of tasks n, total utilization U, hyperperiod H
Ensure: Task set $\mathcal{T} = \{(C_i, T_i)\}_{i=1}^{n}$ such that $\sum_i \frac{C_i}{T_i} \approx U$
1: $U_1, U_2, \ldots, U_n \leftarrow \textsc{RandFixedSum}(n, U)$ ▷ n utilizations summing to U
2: $P \leftarrow$ list of divisors of H such that each $p \in P$ and $p > 10$
3: $\mathcal{T} \leftarrow \emptyset$ ▷ Initialize task set
4: **for** $i = 1$ to n **do**
5: $\quad C_i \leftarrow$ random integer in $[1, 50]$ ▷ Execution time
6: $\quad T_i \leftarrow$ element in P that minimizes $|U_i - \frac{C_i}{T}|$
7: $\quad$ Add task (C_i, T_i) to $\mathcal{T}$
8: **end for**
9: **return** $\mathcal{T}$

3.4.2 Exhaustive test

To investigate potential improvements to traditional partitioning and priority assignments algorithms, all possible partitions would have to be simulated for every taskset, which is practically impossible for task sets of moderate size. However, doing this for smaller task sets may give an indication as to what improvement potential between traditional techniques and the "optimal solution´´. The following test was designed for testing a potential partitioning algorithm X:

For a given taskset size n and number of cores m, consider *all* possible partitions and calculate the considered metric for all partitions and compare the results to X. If there is a large discrepancy between the best performing partitions and X, there

is room for improvement. All metric results are then to be normalized by the best performing partition, effectively converting all other results to a percentage of how close a particular solution were to the “optimal solution´´.

For testing the potential of a priority assignment Y, a similar test was designed. Instead of simulating all partitions over m cores, the simulations were performed on a uni-core machine, trying all possible $n!$ priority assignments.

3.5 Metrics

The following metrics will be used to quantify the security of the system:

- Horizontal schedule entropy: Indicates if schedule entropy is low on any single core, which may be a weak point in the system.
- Vertical schedule entropy: Indicates if a specific group of tasks regularly execute simultaneously on the system.
- Horizontal anterior security: Indicates the average maximum risk of any trusted task to experience an anterior attack from an untrusted task running on the same core.
- Horizontal posterior security: Indicates the average maximum risk of any trusted task to experience a posterior attack from an untrusted task running on the same core.
- Horizontal pincer security: Indicates the average maximum risk of any trusted task to experience a pincer attack from an untrusted task running on the same core.
- Vertical anterior security: Indicates the average maximum risk of any trusted task to experience an anterior attack from an untrusted task running anywhere on the system.
- Vertical posterior security: Indicates the average maximum risk of any trusted task to experience a posterior attack from an untrusted task running anywhere on the system.
- Vertical pincer security: Indicates the average maximum risk of any trusted task to experience a pincer attack from an untrusted task running anywhere on the system.

4

Results

This section presents the results of our simulation-based evaluation of different scheduling strategies and security strengthening techniques. The main metrics of interest are **schedule entropy** and **attack resiliency**, particularly in horizontal and vertical attacker models. Each figure illustrates key performance aspects under varying system configurations.

4.1 Partition Algorithms and Priority Orderings

4.1.1 Exhaustive test

Figure 4.1 illustrates the impact of different priority orderings on schedule entropy on single-core processor. All permutations of priority assignments of 6-task sets were evaluated of task sets for total utilization levels ranging from $U = 0.02$ to $U = 0.8$. For every task set, the results were normalized according to the highest performing priority assignment, to showcase how different priority heuristics perform on average compared to the optimal priority assignment.

Priority orderings that result in shorter response times, such as Rate-Monotonic (RM), provide shorter response time and therefore higher budgets, more flexibility for randomizing task execution, resulting in higher entropy. In contrast, orderings like Increasing Utilization (IU) often lead to longer response times, reducing flexibility and lowering entropy.

Note that with jitter, RM is no longer guaranteed to be the optimal priority ordering. During simulation, there were some high utilization task sets that were not schedulable under RM, but succeeded using another priority assignment generated using OPA.

For some strategies such as IU, very few task sets will be schedulable. In these cases, the Optimal Priority Assignment (OPA) algorithm is used while favoring the desired priority structure, e.g., assigning low-utilization tasks to higher priority levels in the IU configuration. As utilization increases, feasible priority permutations become rare, forcing OPA to generate the same priority ordering regardless of the desired heuristics, causing performance differences to converge.

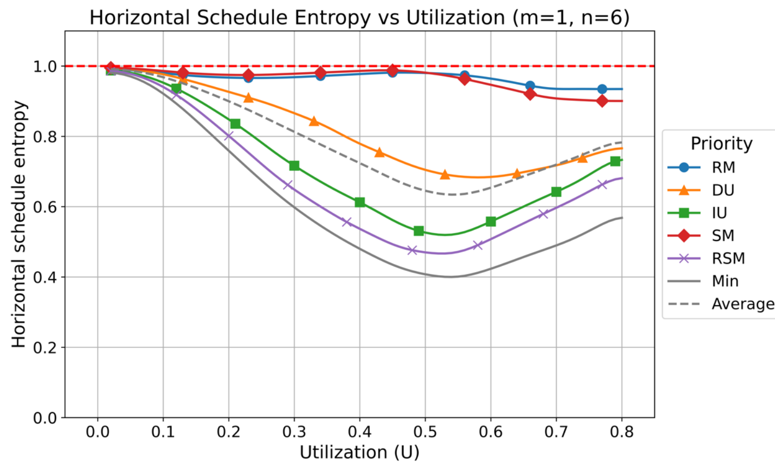


Figure 4.1: Average horizontal schedule entropy across different priority orderings compared to the optimal ordering.

When evaluating the improvement potential for partitioning algorithms, all possible partitions of 5 tasks on 3 cores were evaluated for total utilization levels ranging from $U = 0.05$ to $U = 2.4$. For every task set, the results were normalized according to the highest performing partition, to showcase how different partitioning algorithm performed compared to the optimal partition.

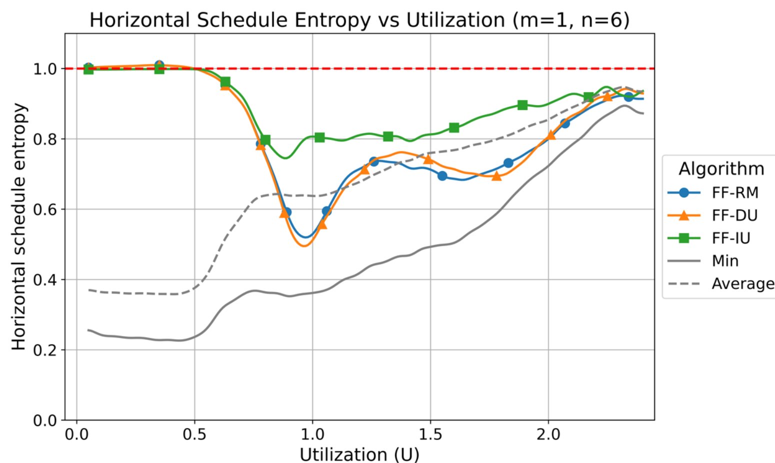


Figure 4.2: Horizontal schedule entropy for First-Fit partitioning under different task picking orders.

Figure 4.2 (FF) and 4.3 (WF) compares how the task allocation strategies affect horizontal schedule entropy at different task set utilization levels. We can immediately tell that for low utilization task sets a high horizontal entropy score is best achieved by allocating all tasks to the same core (FF), rather than WF. However, when the task set utilization approaches 1, First-Fit becomes less favorable as allocating all tasks to one core may lead to high constraints and low inversion budgets. Notably, FF-IU performs well across the entire utilization range. When allocating lower

utilization tasks first, a high utilization task will be the one that might not fit on the initially chosen core, being forced to run by itself on another core. If one task is to run by itself on a core, a medium utilization task would evidently result in a higher single core schedule entropy than a low utilization task, which explains why FF-IU is doing so well. Single-core schedule entropy is maximized at around 55% [5], why it is advantageous to always put tasks on the same core when the task set utilization is below 55%, if the goal is to maximize horizontal schedule entropy.

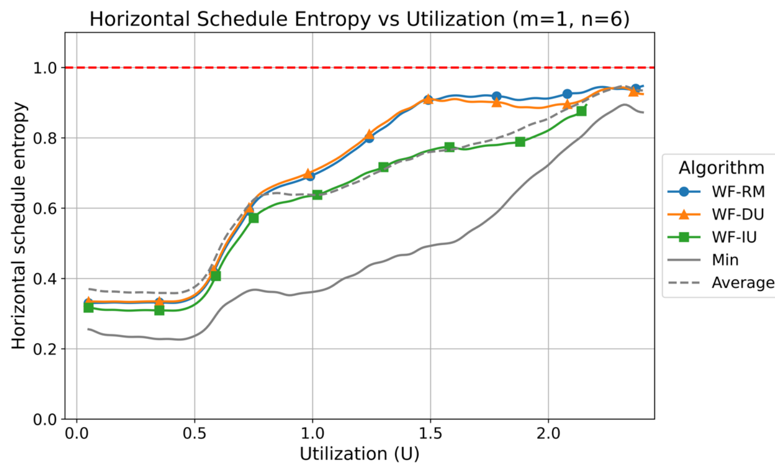


Figure 4.3: Horizontal schedule entropy for Worst-Fit partitioning under different task picking orders.

Figure 4.4 shows the vertical schedule entropy performance for First-Fit partitioning. For low utilization levels, First-Fit resulted in the lowest scores of all possible partitions. This is because FF is likely to leave cores empty, running idle 100% of the time. To achieve high vertical schedule entropy, the probabilities of tasks, and idle tasks, to appear anywhere on the system need to be evenly distributed. If 2 cores are empty, several tasks (namely the idle tasks of the empty cores) always has a 100% chance of running, in this case lowering the vertical schedule entropy of the system. Empty cores are allowed to have a negative impact on vertical schedule entropy because unused resources is a potential opportunity to run two tasks simultaneously, which would be able to happen if only one core is used for instance.

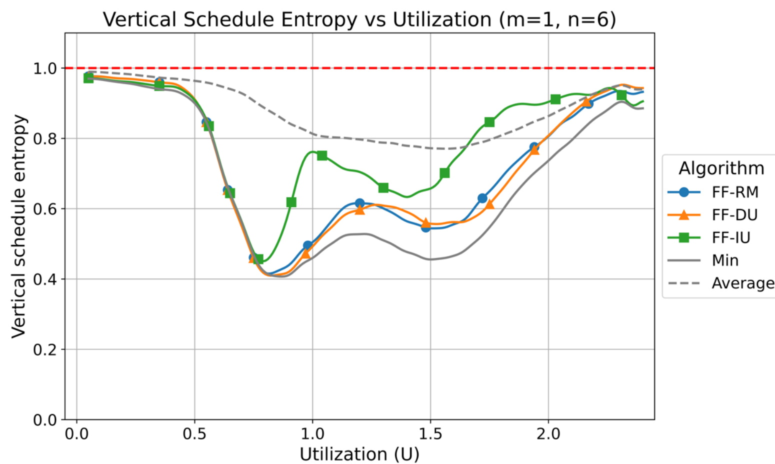


Figure 4.4: Vertical schedule entropy for First-Fit partitioning under different task picking orders.

In contrast, Figure 4.5 shows Worst-Fit achieving much higher vertical entropy scores than First-Fit. Here, the WF-DU strategy (Worst-Fit with Decreasing Utilization ordering) performs near-optimally across all utilization levels. By evenly spreading tasks across cores, this strategy maximizes each core's ability to independently shuffle its tasks, leading to high vertical entropy scores.

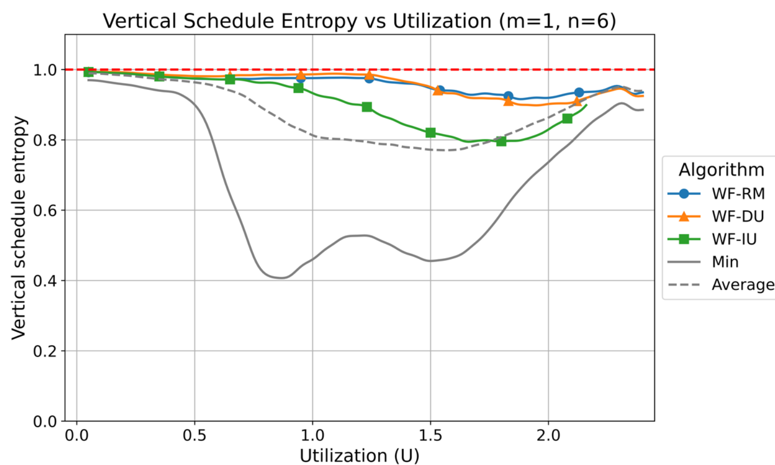


Figure 4.5: Vertical schedule entropy for Worst-Fit partitioning under different task picking orders.

Key takeaways:

- RM priority ordering is close to optimal with regards to schedule entropy, probably because of the resulting low response times, which in turn implies more inversion budget to incorporate randomness.
- Horizontal schedule entropy is generally maximized when every used core has a total utilization as close to 55% as possible. Using fewer cores than what are available is sometimes necessary to achieve this.

- Vertical schedule entropy is generally maximized by an even utilization distribution across cores.

4.1.2 Custom Partitioning Algorithms

Building on insights from earlier experiments, a custom partitioning algorithm was developed in an attempt to simultaneously achieve high horizontal and vertical schedule entropy simultaneously.

WF-minm aims to use the fewest number of cores possible while still achieving a feasible partitioning, depicted in algorithm 3. At line 1, it starts by attempting to allocate the task set on $m = k$ cores, where $k = \lceil U \rceil$, the lowest number of cores physically needed to accommodate a task set of size U. If successful, the algorithms returns this partition at line 4. If unsuccessful, it increases the core count incrementally (to $m = k + 1$, $k + 2$, etc.) until a valid allocation is found (line 8). If the task set could not be partitioned using $m = M$ cores, where M is the number of available cores, the algorithm, reports failure (line 6). In every step, the algorithm uses the WF-RM heuristic to evenly distribute tasks while maintaining medium utilization across the utilized cores. This strategy encourages compaction while avoiding overload.

Algorithm 3 WF-minm

Require: Task set $\mathcal{T}$, minimum number of cores $k = \lceil U \rceil$, available cores M

```
1: m ← k
2: while true do
3:     partition ← WF-RM(T, m)
4:     if partition is feasible then return partition
5:     end if
6:     if m = M then return failure
7:     end if
8:     m ← m + 1
9: end while
```

WF-minm2 adds a utilization cap per core. This is done to prevent cores from saturating, as it has been shown to decrease both horizontal and vertical entropy, see Figure 4.2 and 4.4. Even if all tasks can be feasibly scheduled on m cores, the algorithm chooses to add an extra core if any single core exceeds 60% utilization. This introduces an entropy-aware trade-off: by limiting core saturation, response times are shortened, allowing more headroom for task shuffling, thereby enhancing schedule entropy. A utilization cap of 60% is chosen at this is the point where single-core schedule entropy starts to decrease.

WF-minm2 is presented in algorithm 4. It is similar to algorithm 3, but adds an extra if-clause at line 5 to protect cores from saturating. If it is possible to add more cores to keep core utilization below $U_{\max}$, the algorithm will continue its search.

Algorithm 4 WF-minm2

Require: Task set $\mathcal{T}$, minimum number of cores $k = \lceil U \rceil$, utilization cap $U_{\max} = 0.6$
1: $m \leftarrow k$
2: **while** true **do**
3: $\quad$ *partition* $\leftarrow$ WF-DU($\mathcal{T}, m$)
4: $\quad$ **if** *partition* is feasible **then**
5: $\quad\quad$ **if** core utilization $\leq U_{\max} \forall$ cores OR $m = M$ **then return** *partition*
6: $\quad\quad$ **end if**
7: $\quad$ **end if**
8: $\quad$ **if** $m = M$ **then return** $failure$
9: $\quad$ **end if**
10: $\quad$ $m \leftarrow m + 1$
11: **end while**

WF-minm3 is the same as WF-minm2 but increases $U_{\max}$ slightly, choosing to add an extra core if any single core exceeds 70% utilization instead of 60%. This algorithm exists as an indicator to see how the utilization cap affects the behavior of WF-minm2.

Figure 4.6 and 4.7 shows these strategies performs compares to an exhaustive search of all partitions with regards to horizontal and vertical schedule entropy.

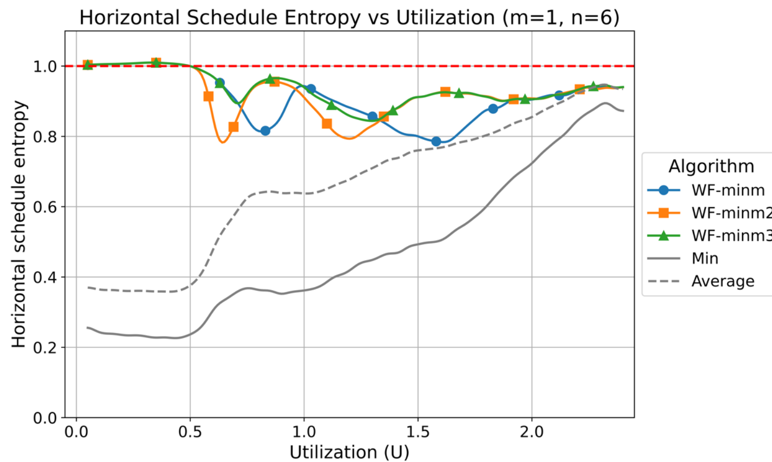


Figure 4.6: Horizontal schedule entropy for custom partitioning algorithms, compared to the optimal partition.

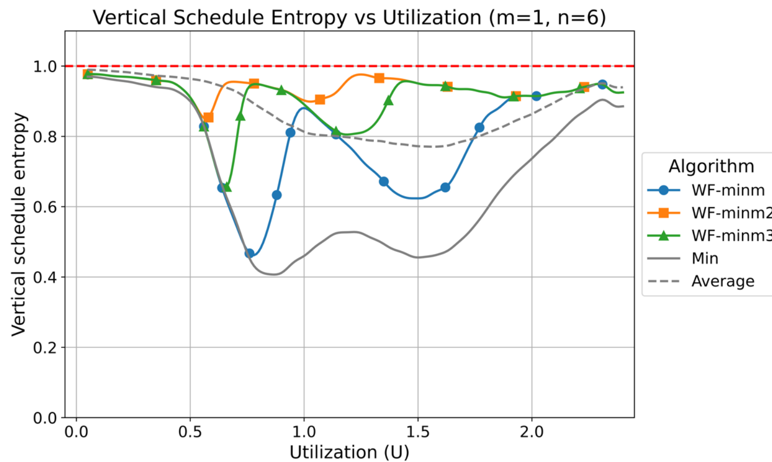


Figure 4.7: Vertical schedule entropy for custom partitioning algorithms, compared to the optimal partition.

Notably, WF-minm2 appears to strike a balance between ensuring high entropy on each core individually as well as across the system as a whole. When increasing the threshold for utilizing an extra core from 60% to 70%, it does more harm than good, as single-core schedule entropy tends to decrease after around 55% [5].

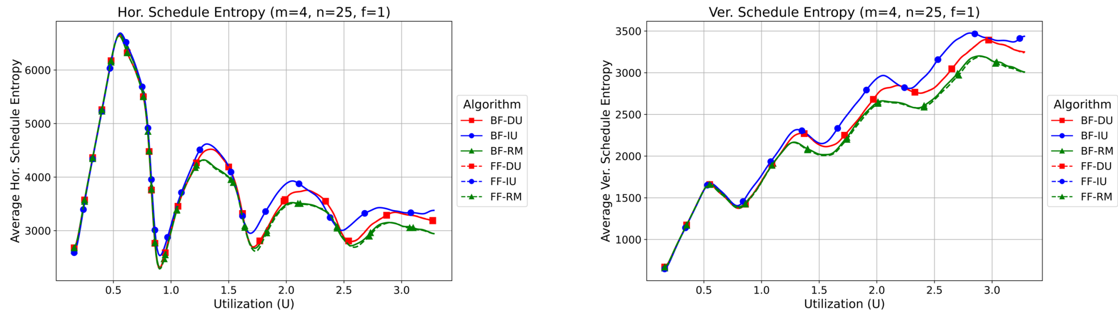
Key takeaways:

- Allocating tasks to less cores than what is available only achieves high schedule entropy scores if a utilization cap is used. This ensures that cores are not saturated if not absolutely necessary.
- The sweet spot for such a utilization cap seems to be at around 60%.

4.1.3 Large scale test: Comparative Analysis of Allocation Algorithms

Figure 4.8 illustrates that **Best-Fit (BF)** and **First-Fit (FF)** yield nearly identical results in terms of schedule entropy. However, the task selection order noticeably impacts entropy levels. Specifically, using a Rate-Monotonic (RM) ordering results in slightly lower schedule entropy compared to other heuristics. This suggests that grouping low-period tasks on the same core can reduce flexibility under the TaskShuffler scheduler.

Among the task picking strategies, the **Increasing Utilization (IU)** heuristic performs the best. This is likely due to its tendency to balance core utilization, as when all tasks do not fit on the same core it will be the higher utilization tasks being moved to an empty core rather than the low utilization ones. Grouping only a few low utilization tasks on the additional core would result in low schedule entropy.

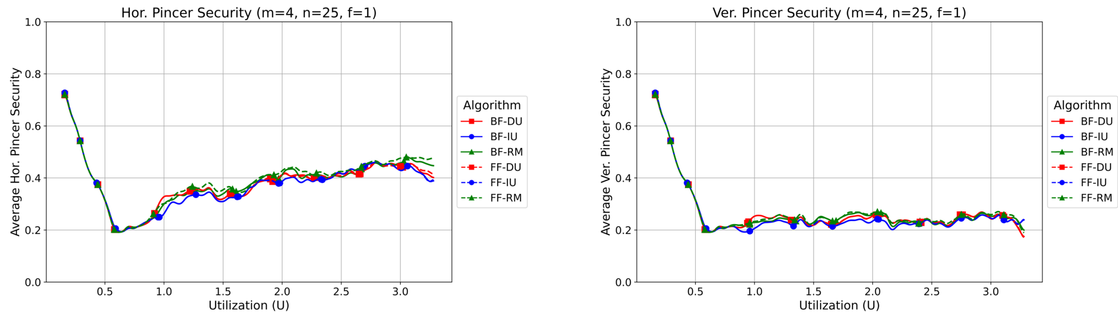


(a) Horizontal entropy peaks at $U = 0.55$ and exhibits periodic dips around $U = kU_{\max}$, where $U_{\max} \approx 0.85$ is the average threshold where task sets typically saturate a core.

(b) Vertical entropy also follows a periodic pattern. Entropy decreases when core utilization peaks and recovers when new cores begin receiving tasks.

Figure 4.8: Schedule entropy for BF and FF algorithms under different task picking orders.

In terms of **attack resiliency**, all BF/FF configurations behave similarly, as shown in Figure 4.9. The choice of task picking heuristic has minimal effect on either horizontal or vertical attack resiliency.



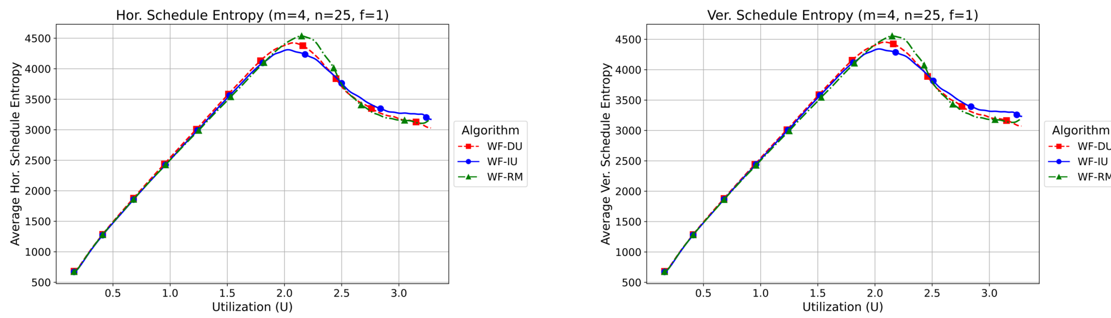
(a) Horizontal anterior attack security for BF and FF.

(b) Vertical anterior attack security for BF and FF.

Figure 4.9: BF and FF show similar anterior attack resiliency regardless of task picking order.

In contrast, the **Worst-Fit (WF)** algorithm distributes tasks evenly across cores, leading to steadily increasing entropy until core capacities are approached. This behavior, depicted in Figure 4.10, is more stable and less periodic than that of BF/FF.

Interestingly, WF-RM performs best in terms of entropy among the WF variants. This is likely because RM spreads low-period tasks across different cores, avoiding concentrated execution patterns and thereby increasing entropy.

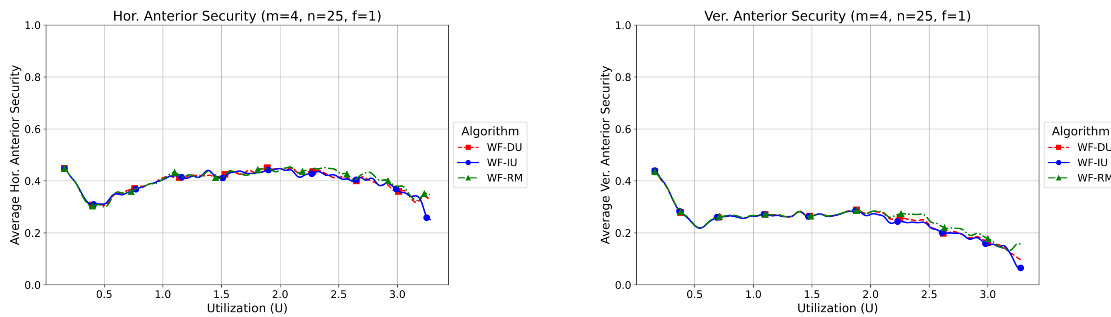


(a) Horizontal entropy increases monotonically as WF balances core loads.

(b) Vertical entropy follows a similar trend with minor dips as cores near capacity.

Figure 4.10: Schedule entropy of WF under different task picking orders.

WF also shows minimal variation in attack resiliency across task picking orders, as indicated in Figure 4.11.



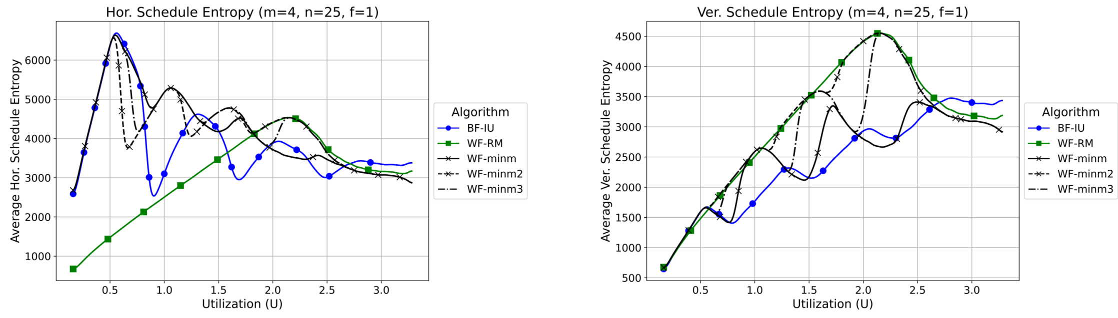
(a) Horizontal anterior attack security.

(b) Vertical anterior attack security.

Figure 4.11: Task picking order has low influence on anterior attack security in WF.

Figure 4.12 compares custom task allocation strategies against the best BF/FF and WF variants (BF-IU and WF-RM, respectively). The custom algorithms generally outperform traditional ones in both horizontal and vertical entropy.

Notably, **WF-minm2** explicitly limits core utilization to 60% when possible, prompting earlier allocation to new cores. This strategy avoids the sharp entropy drop associated with cores reaching full capacity.

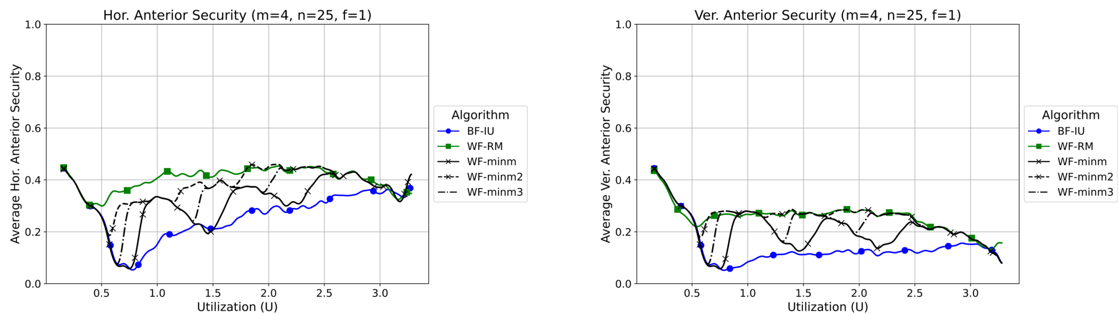


(a) Evenly allocating tasks among a minimal number of cores increases horizontal entropy and avoids drops due to skewed distributions.

(b) WF-minm2 caps core utilization at 60%, enabling better entropy preservation through early core activation.

Figure 4.12: Comparison of schedule entropy across traditional and custom allocation strategies.

Custom algorithms also strike a balance in terms of **anterior attack safety**, as shown in Figure 4.13. BF tends to cluster tasks tightly, increasing risk, while WF spreads them out, reducing attack probability. The custom strategies offer a compromise: packing efficiently while encouraging even utilization across cores.

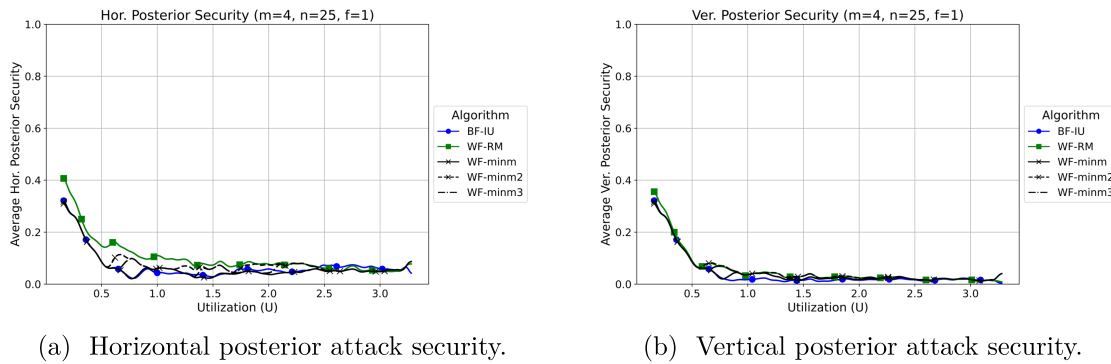


(a) Horizontal anterior attack security.

(b) Vertical anterior attack security.

Figure 4.13: Anterior attack security is highest for WF, lowest for BF. Custom strategies fall in between, balancing compactness and spread.

Finally, Figure 4.14 shows that **posterior attack safety** remains largely unaffected by the choice of allocation algorithm. This suggests that posterior vulnerabilities are more dependent on execution timing and less on core-level allocation strategies.



(a) Horizontal posterior attack security.

(b) Vertical posterior attack security.

Figure 4.14: Posterior attack risk shows minimal dependence on allocation algorithm.

Key takeaways:

- Spreading tasks evenly across all available cores increases attack resiliency.
- Grouping tasks tightly on as few cores as possible results in lower attack resiliency.
- The custom partitioning methods, using few cores but employing a utilization cap strikes a balance between these to extremes, with results somewhere between WF and BF.

4.2 Mitigation strategies

In this section, the different mitigation strategies are first evaluated separately to understand their impact on system security. Later, these strategies are combined.

4.2.1 RePri - Reprioritizing tasks

To evaluate the effect of dynamic reprioritizing during execution, the frequency at which tasks were reprioritized was varied, testing its effect at different set values: 100, 500 and 3000. As shown in Figure 4.15, reprioritizing has a generally negative impact on schedule entropy, particularly in the medium to high utilization range. The default priority assignment used is Rate-Monotonic (RM), which was previously identified as close to optimal with respect to schedule entropy (see Section 4.1.1). Any deviation from RM, *especially* random priority assignments, tends to reduce response times resulting in more deterministic schedule and thus lower schedule entropies.

The figure also shows no substantial difference in entropy between the different reprioritization intervals. This suggests that the act of reprioritization itself, rather than its frequency, is the dominant factor. However, as expected, more frequent reprioritization (shorter intervals) leads to slightly greater reductions in entropy.

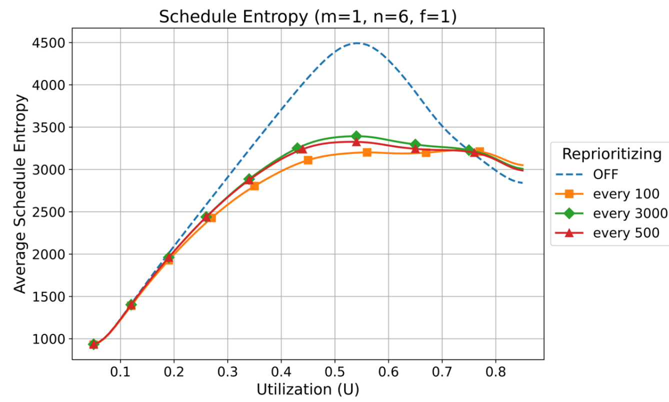
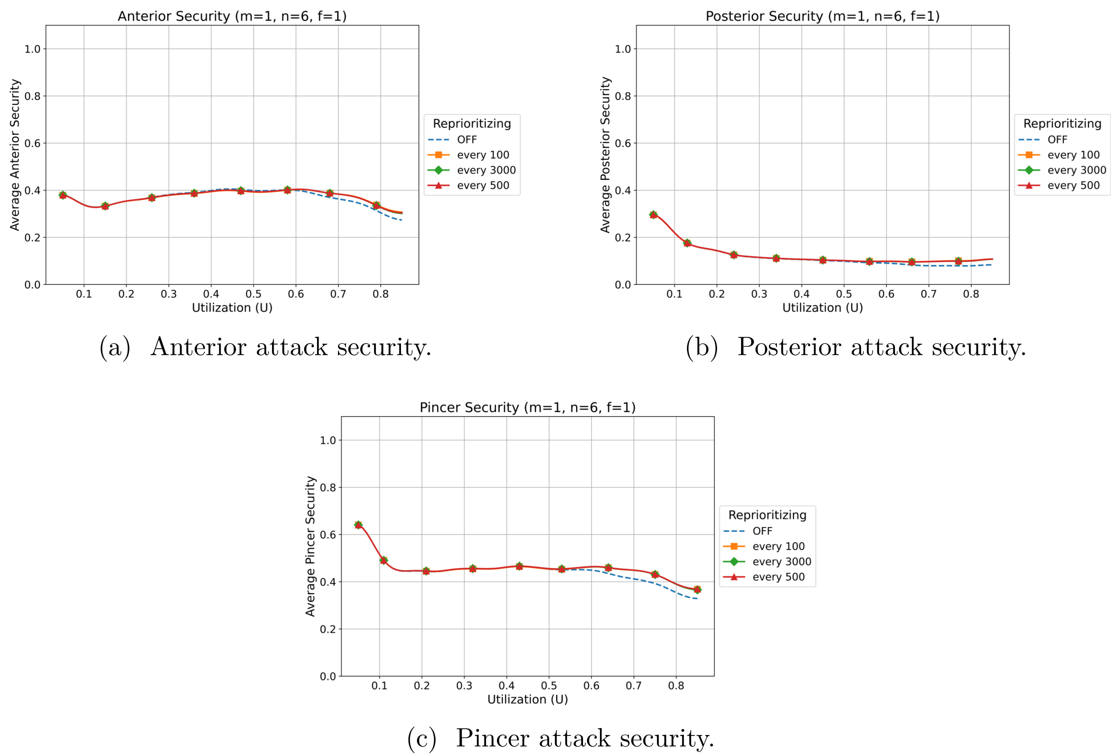


Figure 4.15: Deviating from RM priority ordering reduces schedule entropy, with more frequent reprioritization yielding slightly greater losses.

In terms of attack resiliency, the impact of reprioritization is limited. As shown in Figure 4.16, changes in the reprioritization interval do not significantly affect the likelihood of anterior, posterior, or pincer attacks. The security profiles remain largely unchanged, with only slight variations observed at higher utilization levels.



(a) Anterior attack security.

(b) Posterior attack security.

(c) Pincer attack security.

Figure 4.16: Reprioritization has only a minor effect on attack risk, with noticeable differences limited to high-utilization task sets.

In all subsequent experiments, whenever reprioritization is applied, it is performed once per hyperperiod. This design choice offers two advantages: it enables response-

time tests to be computed more easily and allows priority assignments to be precomputed offline, since the state of all tasks is known at the hyperperiod boundary.

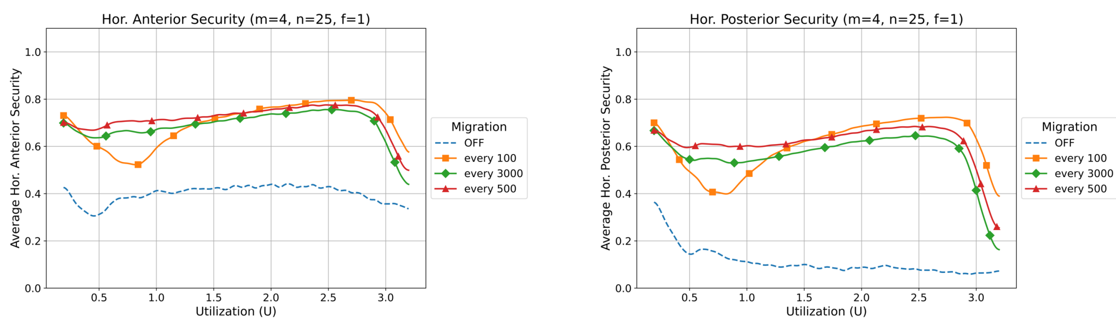
Key takeaways:

- By only using regular reprioritizing one a uni-core processor, the schedule entropy is shown to drop substantially, while having almost zero impact on attack resiliency.

4.2.2 Migration

In a manner similar to reprioritization, the effect of migrating tasks during execution was evaluated by varying the frequency at which migration attempts occurred.

A significant improvement in horizontal attack safety is immediately observed, seen in Figure 4.17. Since tasks can only launch attacks against others on the same core, it is unsurprising that regularly relocating tasks between cores reduces their exposure to such threats.

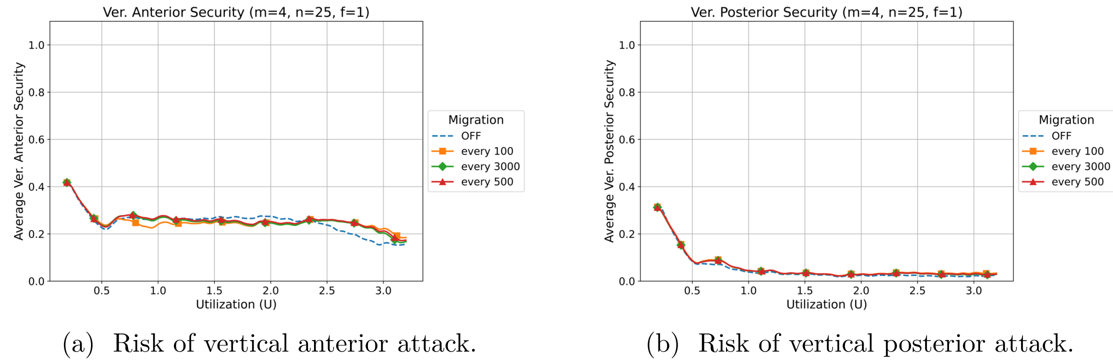


(a) Horizontal anterior attack security.

(b) Horizontal posterior attack security.

Figure 4.17: Regular task migration significantly reduces the risk of horizontal attacks.

Figure 4.18 shows the impact of migration on system-wide (vertical) attack resiliency. If tasks are capable of launching attacks across cores, such as by manipulating shared resources, the benefits of migration are minimal. In such scenarios, migration has little influence on the relative timing between tasks and thus fails to disrupt the conditions that enable these attacks.



(a) Risk of vertical anterior attack.

(b) Risk of vertical posterior attack.

Figure 4.18: Migration has limited impact on system-wide attacks that span across cores.

Migration has a strong influence on horizontal schedule entropy, as illustrated in Figure 4.19a. In these simulations, tasks were allocated using WF-DU. The increase in entropy is primarily due to a rise in the number of distinct tasks running on each core over time. The optimal entropy of a core executing n tasks is given by:

$$H_\Gamma(S_t) = \sum_n -p \cdot \log_2(p) = \sum_n -\frac{1}{n} \cdot \log_2\left(\frac{1}{n}\right) = \log_2(n)$$

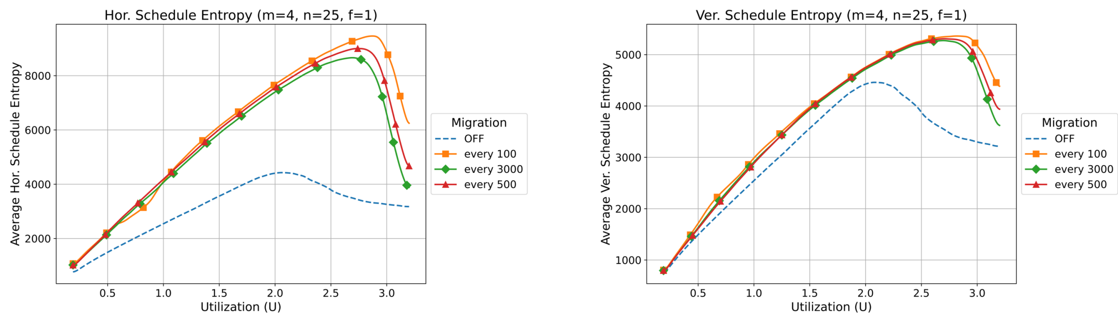
Thus, exposing a core to 25 tasks instead of the average 6.25 increases the entropy by a factor of

$$\frac{\log_2(25)}{\log_2(6.25)} \approx 1.76,$$

which aligns well with the observed results for low- to medium-utilization task sets.

At high utilization levels, cores operate near capacity, limiting TaskShuffler's flexibility. Nonetheless, task migration enables execution in previously unavailable time slots, which increases both horizontal and vertical schedule entropy, as seen in Figure 4.19b. When utilization approaches $U = 3.2 = m \cdot 0.8$, very few task partitions are feasible. Consequently, migrations are less likely to succeed, reducing their effect on entropy. In such cases, simulations with shorter migration intervals show greater impact, since they allow more migration attempts and thus more opportunities to improve entropy.

As the length of the task periods are comparable to the task set hyperperiod (3000), it is reasonable that even just a single migration at every hyperperiod boundary is enough to increase security in such a dramatic way. If the task set had a much longer hyperperiod (300000), migrating once every hyperperiod would not be enough, because the frequency of migration would need to be comparable to the arrival or period of new tasks.



(a) Migration significantly increases horizontal entropy, especially under WF-DU allocation.

(b) Migration has limited effect on vertical entropy, except under high utilization where gains persist.

Figure 4.19: Effect of task migration on schedule entropy. Tasks allocated using WF-DU.

As with reprioritization, task migration, when enabled as a mitigation strategy, is performed once per hyperperiod in the remaining experiments. This approach allows response-time tests to be computed more efficiently, while also enabling partitions to be computed and validated offline, since the full system state is known at each hyperperiod boundary.

Key takeaways:

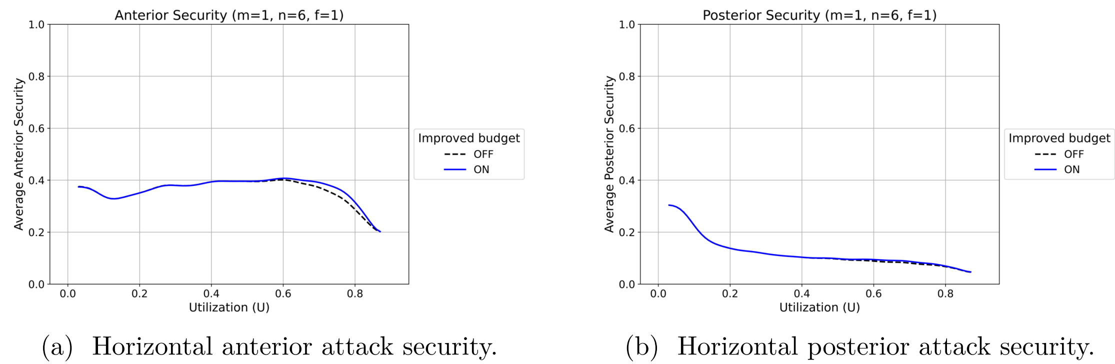
- Migration has huge impact on horizontal schedule entropy and horizontal attack resiliency.
- The effect on vertical schedule entropy is very restricted in the low to medium utilization range, but shows great potential with increasing entropy in high utilization task sets.
- Vertical attack resiliency is practically unaffected. This metric does registers attacks regardless of which cores they run on. To protect against system-wide attacks, the tasks would need to change how and when they execute in relation to their own arrival and deadline.

4.2.3 Uni-core mitigation

In the two following sections, the remaining mitigation strategy are examined individually and in combination with each other to determine how to best protect both uni-core and multi-core systems, starting with uni-core systems.

4.2.3.1 Budget

Improving the budget calculations in TaskShuffler, presented in detail in 2.2.2, has minimal effect on attack resiliency, as shown in Figure 4.20, which compares attack resiliancy for TaskShuffler with and without the improved budget in use.



(a) Horizontal anterior attack security.

(b) Horizontal posterior attack security.

Figure 4.20: Enhanced budget calculations have negligible impact on attack resiliency in single-core systems.

However, improved budget computations do provide increased inversion budgets for high-utilization task sets. This leads to higher schedule entropy, as illustrated in Figure 4.21.

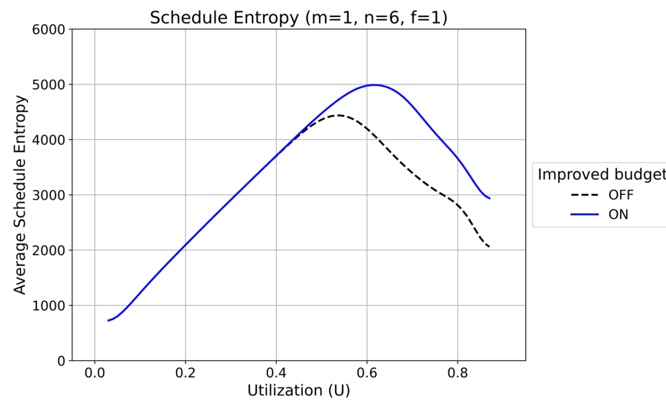


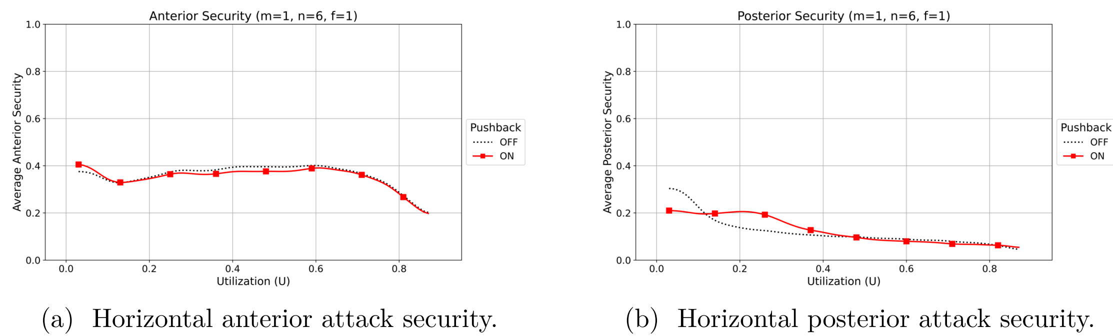
Figure 4.21: Enhanced budget calculations lead to increased schedule entropy for task sets with high utilization.

Key takeaways:

- Improving the budget calculations for taskshuffler is shown to have great impact on schedule entropy, but only for task sets above 50% utilization.

4.2.3.2 Pushback

Pushback was expected to significantly improve posterior attack security. However, its actual impact was minor and, in some cases, even negative, as shown in Figure 4.22.



(a) Horizontal anterior attack security.

(b) Horizontal posterior attack security.

Figure 4.22: Pushback has limited effect on posterior attack security and can, in some cases, reduce it.

In addition, pushback appears to have a detrimental effect on schedule entropy. The mechanism delays task executions toward the end of their deadlines. This behavior introduces predictable patterns where tasks tend to execute near their respective deadline boundaries, thus reducing variability and overall schedule entropy. This is illustrated in Figure 4.23.

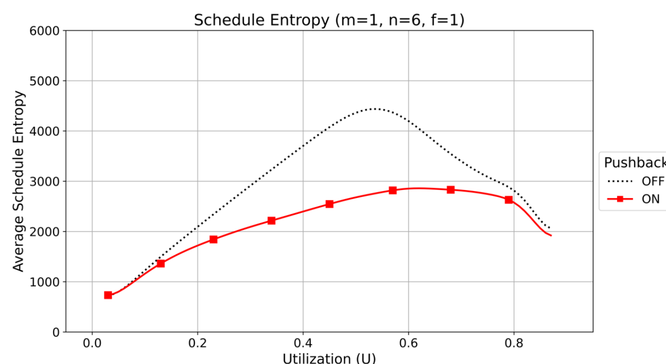


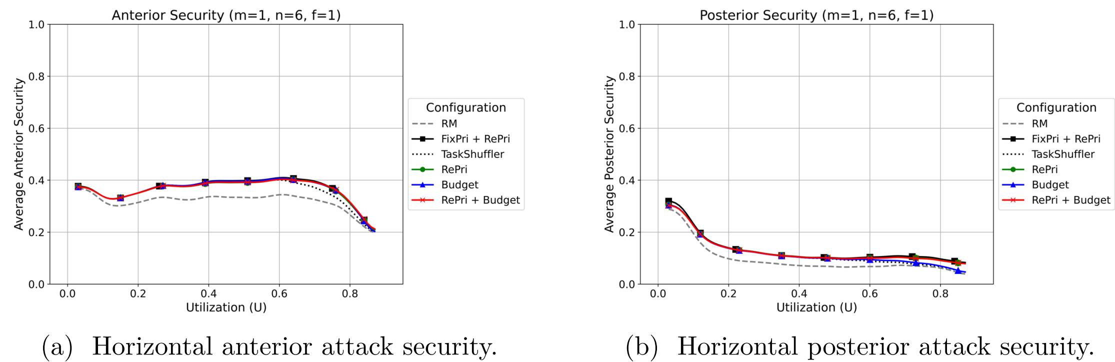
Figure 4.23: Pushback significantly reduces schedule entropy by promoting predictable execution patterns.

Key takeaways:

- Pushback has a clear negative effect on schedule entropy.
- When being used it isolation, its effect on attack resiliency is barely noticeable, and sometimes negative.

4.2.3.3 Mitigation without using pushback

In Figure 4.24, we observe that all combinations of mitigation strategies (excluding pushback) improve both anterior and posterior attack security when compared to the fixed-priority scheduler. Any modification that increases schedule randomness helps reduce the likelihood of recurring attacks.



(a) Horizontal anterior attack security.

(b) Horizontal posterior attack security.

Figure 4.24: Impact of mitigation strategies on attack resiliency.

However, as seen in Figure 4.25, all strategies result in lower schedule entropy compared to TaskShuffler with improved budget calculations. As previously discussed, any priority ordering that deviates from rate-monotonic typically reduces the inversion budget available to tasks, thereby limiting the entropy-enhancing effect of TaskShuffler.

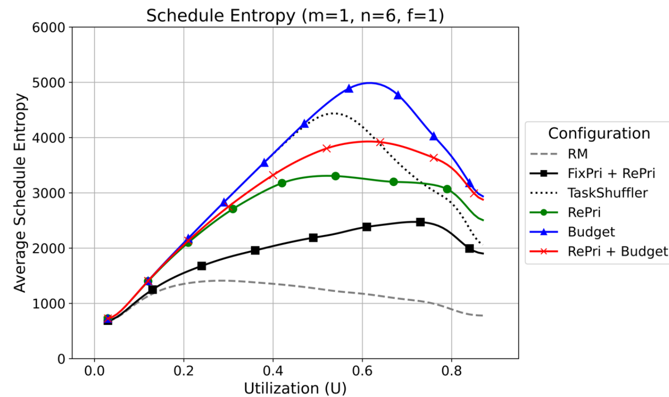


Figure 4.25: Most mitigation strategies reduce schedule entropy compared to budget-optimized TaskShuffler.

Key takeaways:

- Without Pushback, the impact on attack resiliency is minor.
- The improved budget calculations for task shuffler helps mitigate the negative effect reprioritizing has on schedule entropy.

4.2.3.4 Combining all mitigation strategies

A clear improvement in posterior attack security is shown in Figure 4.26. The combination of pushback and reprioritization has a significant impact on posterior security, with a slight additional benefit when improved budget calculations are also applied. Without reprioritization, pushback leads to a more predictable schedule. Even if tasks individually tend to finish closer to their deadlines, a low-priority task

may still be consistently exposed to interference from a high-priority task, forcing it to always finish its execution early. Since the security metric reflects the safety of the *least* secure task, maintaining a dynamic priority ordering is essential to ensure all tasks benefit from pushback.

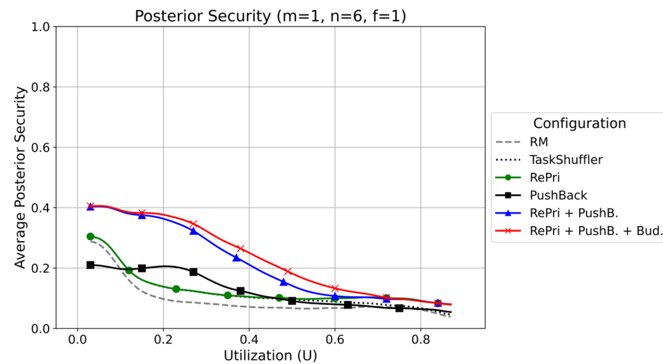
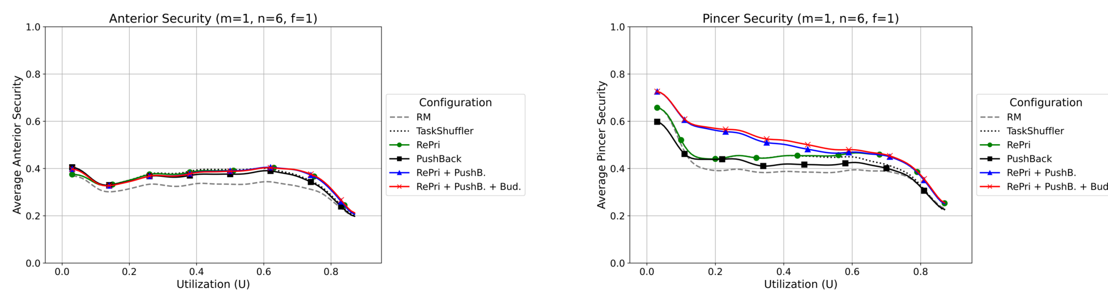


Figure 4.26: Combination of pushback and regular reprioritization significantly improves posterior attack security.

None of the proposed attack mitigation strategies seem to affect the anterior security anymore than taskshuffler already does. The pincer security metric has increased, mainly due to the hightened posterior security score, see Figure 4.27.

None of the proposed mitigation strategies improve anterior attack security beyond what TaskShuffler already achieves. However, the pincer attack security metric increases, primarily due to the improved posterior security score, as shown in Figure 4.27.



(a) Anterior attack security shows minimal change.

(b) Improved pincer security is primarily driven by gains in posterior attack security.

Figure 4.27: Overall improvement in attack resiliency when combining mitigation strategies.

In terms of schedule entropy, the improved budget calculations are not sufficient to counteract the reduction caused by pushback and reprioritization. As shown in Figure 4.28, the resulting schedule is more secure but less entropic than the baseline with only improved budget calculations.

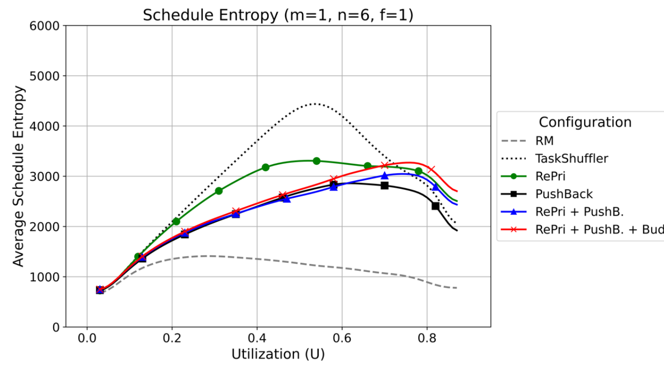


Figure 4.28: Improved budget calculations do not fully offset the entropy loss introduced by other mitigation strategies.

Key takeaways:

- Pushback and Reprioritizing only displays great benefit for posterior attack security when used in combination with each other. They do not, however, noticeably increase anterior security.
- When Pushback and Reprioritizing is used in combinations, the new budget calculations are not as effective mitigating the drop in schedule entropy.

4.2.4 Multi-core mitigation

This section explores how to best protect multi-core systems, when *all* mitigation strategies, including migration, are available.

The greatest improvement in horizontal anterior security comes, unsurprisingly, from migration. This effect is slightly amplified when combined with reprioritization, as shown in Figure 4.29.

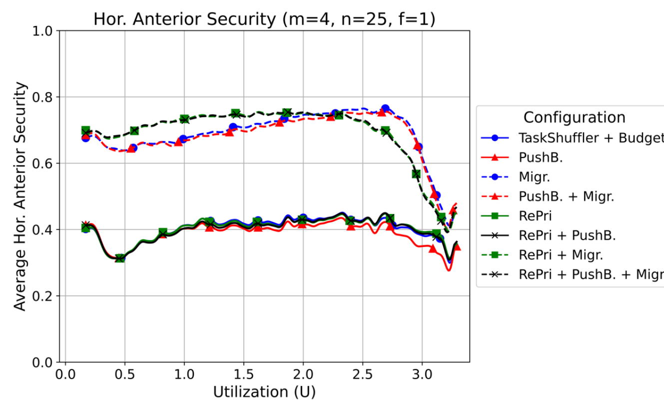


Figure 4.29: Reprioritization improves horizontal anterior security only when used alongside migration.

As illustrated in Figure 4.30, migration once again has a major impact, this time on posterior security. The score improves even further when both pushback and

reprioritization are used in combination. For smaller task sets, pushback and reprioritization alone can nearly match the posterior security benefits of migration. Since migration may incur substantial runtime overhead, it may be beneficial to avoid it when comparable protection can be achieved without it.

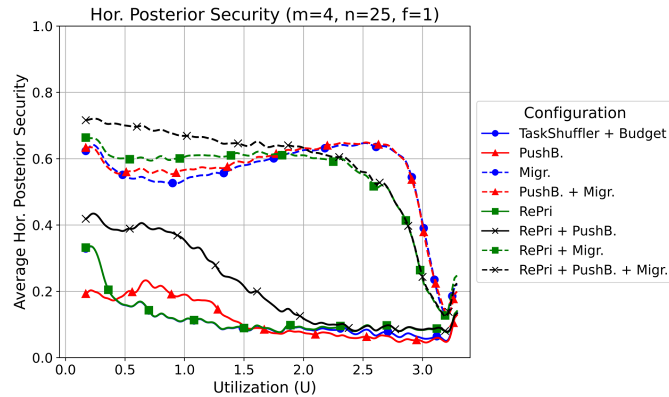


Figure 4.30: Pushback significantly improves horizontal posterior security only when combined with reprioritization.

Migration has only a minor effect on vertical posterior security, as seen in Figure 4.31. In contrast, the combination of pushback and reprioritization yields a more noticeable improvement.

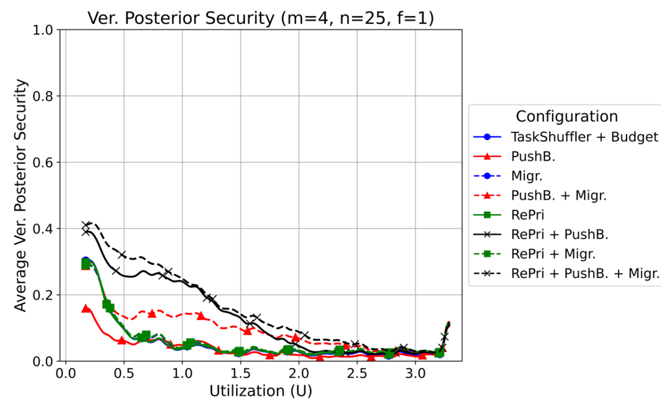
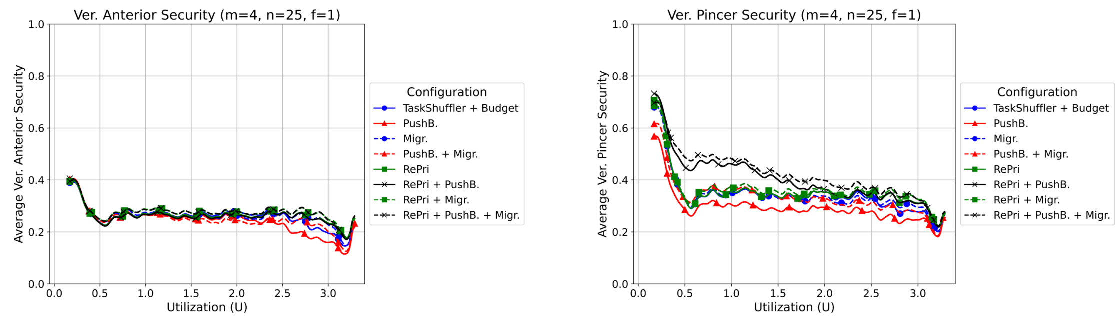


Figure 4.31: Slight improvement in vertical posterior security. Migration contributes minimally.

None of the evaluated mitigation strategies lead to a notable increase in vertical anterior security, as shown in Figure 4.32.

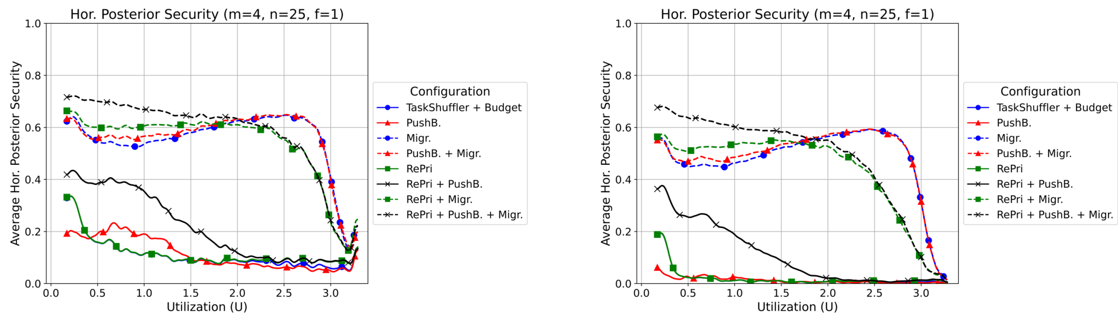


(a) Vertical anterior security remains nearly unchanged.

(b) Vertical pincer security.

Figure 4.32: Mitigation strategies show little to no improvement in vertical anterior security.

As the number of untrusted tasks increases, the risk of attack approaches 100%, causing the security score to drop toward zero. However, the proposed mitigation strategies remain effective even when the number of compromised tasks increases to $f = 3$, as demonstrated in Figure 4.33.

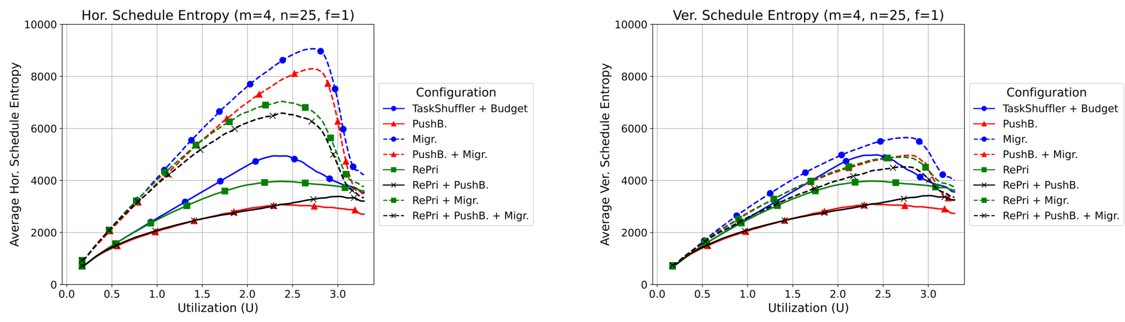


(a) Protection against horizontal posterior attacks when $f = 1$.

(b) Protection against horizontal posterior attacks when $f = 3$.

Figure 4.33: Without mitigation, security rapidly degrades as the number of untrusted tasks increases.

Regarding schedule entropy, migration has a significant positive impact. It helps recover much of the entropy loss caused by other mitigation strategies, as illustrated in Figure 4.34.



(a) Effect of mitigation strategies on horizontal schedule entropy.

(b) Effect of mitigation strategies on vertical schedule entropy.

Figure 4.34: Multi-core mitigation strategies mitigate the drop in schedule entropy.

Key takeaways:

- Migration is a crucial precautionary technique for preventing both anterior and posterior horizontal attacks.
- Not even migration, nor any other security strengthening strategy increases vertical anterior security.

5

Conclusion

This chapter summarizes the findings of this report, with a focus on deriving practical recommendations for defending real-time systems against different classes of attackers. The discussion highlights key patterns in the data and the effectiveness of mitigation strategies across various security metrics. Finally, potential directions for future work are outlined.

5.1 Discussion

The results clearly show that the effectiveness of the proposed protection strategies depends significantly on the attacker's position (anterior or posterior) and scope (horizontal or vertical). A nuanced approach is therefore required to secure systems in practice.

Posterior Attack Protection

Among all strategies, posterior attack mitigation saw the most significant gains. The combination of *pushback* and *reprioritization* proved highly effective for both horizontal and vertical posterior threats. Pushback on its own creates a more predictable schedule, so without reprioritization, the benefits are unevenly distributed. Some low-priority tasks remain persistently vulnerable. Reprioritization ensures that all tasks can benefit from pushback by rotating their relative positions over time.

When migration is introduced in multi-core systems, posterior security improves dramatically. Notably, even pushback and reprioritization without migration can achieve almost comparable results for low utilization task sets. This suggests that migration, while powerful, might not always be necessary. This is especially useful when the overhead it might introduce is a dictating factor.

Anterior Attack Protection

Unfortunately, anterior attacks remain difficult to mitigate. None of the proposed strategies, including pushback, reprioritization, or even migration, led to significant improvements in vertical anterior security. Horizontal anterior security benefited substantially from migration, particularly when paired with reprioritization.

Pincer Attack Protection

As for pincer attacks, the improvements observed largely stem from gains in posterior security. This makes sense, as pincer attacks require both anterior and posterior exposure. Protecting the system from even one of these is often enough to significantly reduce the overall risk.

Schedule Entropy

There is an inherent trade-off between attack resiliency and schedule entropy. Reprioritization and pushback reduce entropy, making the schedule more predictable and thus less flexible. However, **migration appears to be a game-changer** in this regard. It mitigates the entropy loss from other strategies, especially in multi-core settings. This makes migration not only a security enhancement but also a stabilizing force in terms of maintaining runtime diversity.

5.2 Conclusion

The following key insights emerge from this study:

- **Migration is highly effective**, particularly in multi-core systems. It boosts both security and schedule entropy, and works well in combination with other strategies.
- **Pushback and reprioritization are critical** for defending against posterior attacks. Together, they greatly reduce the exposure of the least secure tasks.
- **Anterior attacks require migration to be avoided**. Vertical anterior security is especially difficult to enhance.
- **Attacks resiliency comes at a cost to entropy**, but this trade-off can be mitigated through use of migration.

Recommendations can be made depending on the assumed threat model:

- For systems primarily concerned with **posterior attacks**, adopt a combination of pushback, reprioritization, and, if resources allow, migration.
- For systems with known **horizontal anterior advertsaries**, migration and reprioritization offer substantial protection.
- For systems exposed to **vertical anterior threats**, current methods are insufficient, and specialized solutions are needed. The risk of anterior attacks can however be minimized by choosing task allocation strategies that distributes tasks among cores as evenly as possible.

5.3 Future Work

While the proposed methods offer solid improvements, there is still room for advancement, especially in the following areas:

- **Trade-off between security and efficiency:** This study completely disregards the effect of potential overhead when performing frequent context switches, migration and reprioritization. In practice, attempts to highten security might lead to loss of effective computational power. Finding a balance between these to important goals is important to configure a system that both runs safetly, but more importantly, actually runs.
- **Improving vertical anterior security:** To increase resiliency to anterior attacks, the scheduler could promote newly arrived task to execute on the core as soon as possible, to minimize the probability of other tasks running before its first execution. By fine-tuning how the scheduler handles tasks as they arrive, anterior attacks might be avoided more often, while keeping the other security metrics high.

Bibliography

[1] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An empirical survey-based study into industry practice in real-time systems," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2020, pp. 3–11.

[2] J. P. Cerrolaza, R. Obermaisser, J. Abella, *et al.*, "Multi-core devices for safety-critical systems: A survey," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–38, 2020.

[3] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the ukrainian power grid," SANS Industrial Control Systems, Tech. Rep., 2016.

[4] D. Albright, P. Brannan, and C. Walrond, "Did stuxnet take out 1,000 centrifuges at the natanz enrichment plant?" Institute for Science and International Security, Tech. Rep., 2010.

[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)*, IEEE, 2016, pp. 1–12.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[7] K. Krüger, M. Volp, and G. Fohler, "Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems," *Leibniz International Proceedings in Informatics*, vol. 106, 2018.

[8] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys (CSUR)*, vol. 43, no. 4, pp. 1–44, 2011.

[9] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software engineering journal*, vol. 8, no. 5, pp. 284–292, 1993.

[10] N. C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," Department of Computer Science, University of York, York, U.K., Technical Report YCS 164, 1991.

[11] 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)*, IEEE, 2019, pp. 103–116.

[12] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-time systems*, vol. 30, no. 1, pp. 129–154, 2005.

[13] R. I. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Systems*, vol. 47, pp. 1–40, 2011.

[14] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni, "Beyond just safety: Delay-aware security monitoring for real-time control systems," *ACM Transactions on Cyber-Physical Systems (TCPS)*, vol. 6, no. 3, pp. 1–25, 2022.

[15] J. Ren, Z. Wang, C. Lin, *et al.*, "Reorder++: Enhanced randomized real-time scheduling strategy against side-channel attacks," *IEEE Transactions on Network Science and Engineering*, vol. 10, no. 6, pp. 3253–3266, 2023.

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

[17] 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)*, IEEE, 2023, pp. 32–41.

[18] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, 2010, pp. 6–11.

[19] A. Kerckhoffs, "La cryptographie militaire," *J. Sci. Militaires*, vol. 9, no. 4, pp. 5–38, 1883.